Noura Haji & Vinh-Phu Phan

Erzeugung von Benutzungsschnittstellen auf Basis von Zustandsdiagrammen und Fensterentwürfen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Jörg Raasch
Zweitgutachter : Prof. Dr. rer.nat. Christoph Klauck

Abgegeben am 19. Januar 2007

Noura Haji & Vinh-Phu Phan

Thema der Bachelorarbeit

Erzeugung von Benutzungsschnittstellen auf Basis von Zustandsdiagrammen und Fensterentwürfen

Stichworte

Benutzungsschnittstelle, GUI, Zustandsdiagramme, Prototyping

Kurzzusammenfassung

Dem Entwickler bei der Erstellung von Benutzungsschnittstellen eine angemessene Unterstützung zu bieten, ist ein wesentliches Bestreben dieser Bachelor-Arbeit. Bisher ist die Entwicklung von Benutzungsschnittstellen mit erheblichem Aufwand verbunden;Gründe dafür sind fehlende Methoden und unzureichende Werkzeuge. In dieser Bachelor-Arbeit wird die Methode der Erzeugung von Benutzungsschnittstellen mit Hilfe von Zustandsdiagrammen und Fensterentwürfen vorgestellt. Für die Umsetzung dieser Methode wurde eine Lösung entwickelt, die die Funktionalität eines GUI-Builders mit der eines Zustandsdiagrammeditors zweckmäßig kombiniert und beide Komponenten in geeigneter Wiese in den Softwareentwicklungsprozess integriert. Diese Lösung heißt SUIT. SUIT erzeugt aus einem Modell automatisch Prototypen .Weiterhin wird das Modell in SUIT zur automatischen Codegenerierung genutzt. Prototypen können in SUIT bis zur fertigen Anwendung weiterentwickelt werden.

Ende des Textes

Noura Haji & Vinh Phu Phan

Title of the paper

Developing User-Interface with state diagram and window designs.

Keywords

User-Interface, GUI, state diagram, Prototyping

Abstract

The main object of this bachelor's thesis is to support the developer in building user interfaces. Due to missing methods and insufficient tools the former development effort is extraordinary. This bachelor's thesis presents a method to create user interfaces by means of state diagrams and window designs. The solution to implement this method was to combine the functionality of a GUI-builder with a state diagram editor and to integrate this into the software development process. This solution is called SUIT. SUIT takes a model and creates automatically prototypes and code. The prototypes which are built in SUIT can be developed to running software applications.

End of text

Erklärung

Hiermit versichern wir, dass wir diese Arbeit selbständig verfasst haben, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.

Hamburg, 19. Januar 2007; Vinh Phu Phan
Hamburg, 19. Januar 2007; Noura Haji

Danksagung

An dieser Stelle möchten wir allen danken, die bei der Entstehung dieser Arbeit mitgewirkt haben. Insbesondere bedanken wir uns bei unseren Betreuer Prof. Dr. J. Raasch, der uns mit seinen Ratschlägen und seiner Hilfe immer zur Seite stand.

Ebenso möchten wir uns bei Prof. Dr. rer. nat. C. Klauck bedanken, der als Zweitgutachter uns offen mit Ratschlägen und Anregungen geholfen hat.

Unser Dank richtet sich auch an die freiwilligen Tester, die im Usability-Labor sich an SUIT herangetraut haben und uns mit ihren Meinungen und Kommentaren bzgl. des Tools weitergeholfen haben.

Inhaltsverzeichnis

1. Einleitung 7
   1.1 Motivation 7
   1.2 Vision 8
      1.2.1 Methode 8
      1.2.2 Vision SUIT 13
      1.2.3 SUIT Komponenten 14
      1.2.4 Einsatz SUIT 15
   1.3 Aufbau der Arbeit 16

2. Marktanalyse 17
   2.1 Einleitung 17
   2.2 Grobe Anforderungen 17
   2.3 Existierende Tools 18
   2.4 GUI-Builder 18
      2.4.1 Jigloo 19
      2.4.2 Integrierter GUI-Builder von Microsoft Visual Studio.Net 22
      2.4.3 Folgerung 24
      2.4.4 Fazit 24
   2.5 Zustandsdiagramm-Editoren 25
      2.5.1 Together 6.2 25
      2.5.2 MS Visio 26
      2.5.3 Folgerung 27
   2.6 Fazit 28

3. Theorie 30
   3.1 Einleitung 30
   3.2 Benutzungsschnittstellen auf Basis von endlichen Automaten und Fensterentwürfen 30
   3.3 Vorgehensweise mit Interaktionsdiagrammen 31
      3.3.1 Interaktionsdiagramme 31
      3.3.2 Virtuelle Tasten 33
      3.3.4 IAD- Zustände und Masken 34
      3.3.5 Dialogwechsel 34
      3.3.6 IAD-Aktionen 35
   3.4 Die in dieser Arbeit gewählte Vorgehensweise 37
      3.4.1 Vorgehensweise mit Zustandsdiagrammen 37
      3.4.2 SUIT-Notation 39
   3.5 Fazit 40

4. Anforderungen 41
   4.1 Einleitung 41
   4.2 funktionale Anforderungen 41
   4.3 Fazit 47

5. SUIT-Architektur 48
   5.1 Einleitung 48
   5.2 SUIT-Architektur Zusammenhang mit VS.Net 48
   5.3 Interne SUIT-Architektur 49

5.4 Architektur des Zielsystems 52
5.5 Fazit 53

6. Umsetzung von SUIT 55
6.1 Einleitung 55
6.2 Umgesetzte Anforderungen 55
6.3 Programmstruktur 57
6.4 Besonderheiten 59
6.4.1 Erweiterung der Toolbox um Zustandsdiagramm-Elemente 59
6.4.2 Öffnen mit dem Zustandsdiagramm-Designer in VS.Net 61
6.4.3 Programmiersprachunabhängigkeit in der Codegenerierung in .Net 61
6.4.4 Das Anzeigen von Eigenschaften in einem Eigenschaftsfenster 64
6.4.5 Aufrufen des GUI-Designers in SUIT 65
6.5 Schnittstelle von SUIT 66
6.5.1 ZustandsdiagrammElement 66
6.5.2 ZDiagrammElementDesigner 69
6.6 Fazit 75

7. Fallstudie 76
7.1. Einleitung 76
7.2 Ist-Analyse 76
7.3 Schrittweise Erzeugung des Prototyps 77
7.3.1 Vorgehensweise 77
7.3.2. Identifikation der elementaren Werkzeuge 79
7.3.3 Entwicklung der Benutzungsschnittstelle auf Basis eines Zustandsdiagramms 80
7.3.4 Angebotsverwaltungswerkzeug 80
7.3.5 Kundenverwaltungswerkzeug 83
7.3.6 Bestellverwaltungskombiwerkzeug 86
7.3.7 Integration der Werkzeuge zum Gesamtsystem 87
7.4 Fazit 90

8. Usability 91
8.1. Einleitung 91
8.2 Usability Aufgabe 91
8.3 Testverlauf 91
8.4 Ergebnisse 92
8.4.1 Erste Testperson 93
8.4.2 Zweite Testperson 94
8.4.3 Dritte Testperson 95
8.4.4 Vierte Testperson 96
8.5 Fazit 97

9. Möglichkeiten und Grenzen von SUIT 99
9.1 Einleitung 99
9.2 Möglichkeiten von SUIT 99
9.3 Erläuterung der Grenzen 100
9.3.1 Verteilte Benutzungsschnittstellen 100
9.3.2 Codegenerierung für den Aufruf des Folgezustands 101
9.2.6 Konsistenzverlust in späteren Phasen 104
9.4 Fazit 105

10. Ausblick 106
10.1 Einleitung 106
10.2 Erweiterungen von SUIT 106
10.2.1 Erweiterung Hierarchie 106
10.2.2 Erweiterung Konsistenzerhaltung 107
10.3 Fazit 111

11. Resumé 112

Quellen 113
Abbildungsverzeichnis 114
Tabellenverzeichnis 116
Index 117

Anhang: 120
Benutzerhandbuch 121
Klassendiagramm 139

1. Einleitung

Die Softwaremodellierung besitzt im Software-Entwicklungsprozess einen hohen Stellenwert. Hier hat sich in der objektorientierten Software-Entwicklung UML in wenigen Jahren als Standardsprache durchgesetzt.

UML bietet neun Typen von Diagrammen, doch keins davon geht auf die Erfordernisse beim Entwurf von Benutzungsschnittstellen ein, obwohl noch vor wenigen Jahren eine Studie von Myers und Rosson ergab, dass ca. 50 % des Gesamtaufwandes eines Softwareprojektes alleine für die Programmierung der graphischen Benutzungsschnittstelle notwendig war [MyersRosson92]. Gründe hierfür sind unter anderem fehlende Methoden und unzureichende Werkzeuge.

Ein wesentliches Bestreben dieser Bachelor-Arbeit ist es, den Entwickler bei der Erstellung von Benutzungsschnittstellen angemessen zu unterstützen. Für diesen Zweck wird im Rahmen dieser Bachelor-Arbeit ein Werkzeug für die Erzeugung von Benutzungsschnittstellen auf Basis von Zustandsdiagrammen und Fensterentwürfen realisiert.

1.1 Motivation

Zum Entwurf von Benutzungsschnittstellen ist es ab einer bestimmten Größe unerlässlich, ein Modell ihres dynamischen Verhaltens zu entwickeln. Dafür ist ein geeignetes Tool erforderlich. Viele auf dem Markt angebotenen Modellierungstools dienen allerdings in erster Linie nur der Modellierung. Diese Tools können eher als Graphikprogramme angesehen werden, da sie hauptsächlich den Anwender beim Erstellen von Diagrammen unterstützen. Die Diagramme können aber letztendlich nur gespeichert oder ausgedruckt werden.

Aus Sicht des Programmierers und Software-Entwicklers wäre es jedoch wünschenswert, aus einem formal eindeutigen Modell möglichst viele der niedergelegten Informationen zur automatischen Erzeugung von Code für die Benutzungsschnittstelle weiter zu verwenden. In einigen Fällen können Modelle während der Spezifikation bereits so detailliert beschrieben werden, dass aus ihnen zumindest ein Programmgerüst, wenn nicht sogar ausführbarer Code, automatisch generiert werden könnte.

Weiterhin ist es erstrebenswert, aus einem Modell einen lauffähigen Prototypen zu erstellen, das in den frühen Phasen der Softwareentwicklung zum Einsatz kommen kann. Modelle sollten nicht nur als Vorlage dienen, sondern gewinnbringend für die Weiterentwicklung einer Benutzungsschnittstelle eingesetzt werden. Ziel ist es also die Modellierung von Benutzungsschnittstellen mehr in den Softwareentwicklungsprozeß zu integrieren und damit den Nutzen eines Modells zu steigern.

1.2 Vision

1.2.1 Methode

Beim Entwurf von Benutzungsschnittstellen geht ein Entwickler gewöhnlich so vor, dass er ein Modell ihres Verhaltens entwirft: er skizziert bildhaft die Masken und legt die Reihenfolge ihres Ablaufs fest.

Abb. 1 zeigt ein mögliches Beispiel solch eines Modells:

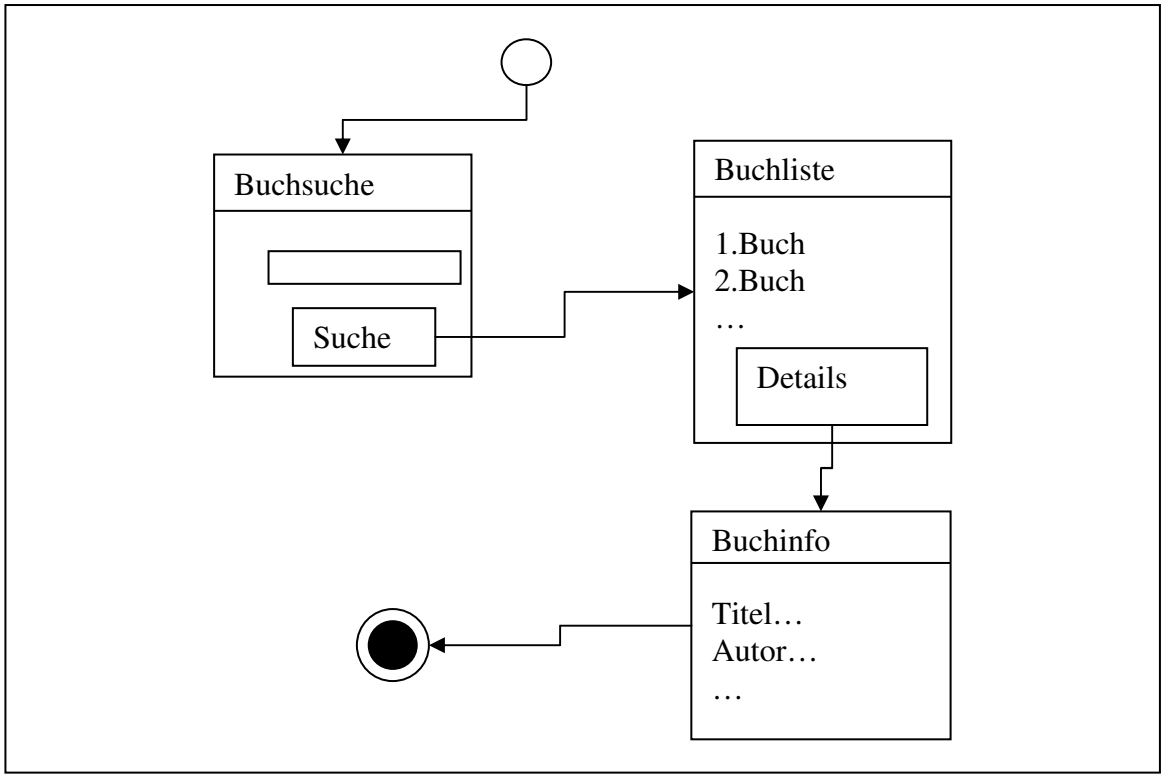


Abb. 1 Literatur-Suche

Abb. 1 beschreibt die Dialogabläufe der Literatur-Suche in Form von Masken. Die Benutzungsschnittstelle befindet sich zu jedem Zeitpunkt in einer von mehreren Masken. Benutzereingaben führen dazu, dass zwischen den Masken hin und her navigiert werden kann. Die Skizze der Maske Buchsuche im Beispiel besitzt ein Eingabefeld, in dem der Suchbegriff eingegeben wird. Klickt man auf den Button „Suche“, so gelangt man in die nächste Maske Buchliste. Hier werden alle zu dem Suchbegriff passenden Bücher aufgelistet. Wird nun ein Buch aus der Liste ausgewählt, wechselt man durch einen Klick auf „Details“ zu der Folgemaske „Buchinfo“, in der die Details des ausgewählten Buches angezeigt werden.

Ein solches Modell wie das beschriebene entsteht intuitiv, allein aufgrund des Versuchs, die Abläufe in einer Benutzungsschnittstelle, also die auftretenden Abfolgen von Masken und die damit verbundenen Eingaben und Ausgaben zu beschreiben: Was durch die obige Abbildung jedoch deutlich wird, ist, dass dieses Modell, aus der Logik der Vorgehensweise heraus, ein bestimmtes Muster, eine Struktur aufweist. Im Modell bilden die Masken die Knoten und die Pfeile, mit denen die Masken verbunden werden, die Kanten. Wird eine Eingabe betätigt, so wird zu der Folgemaske gewechselt. Beim näheren Betrachten des Modells handelt es sich um nichts anderes als einen endlichen Automaten.

Der endliche Automat ist folglich kein Resultat einer besonderen methodischen Vorgehensweise, sondern ein Ergebnis, das während des bildhaften Entwurfs der Dialogabläufe durch eine Anzahl von aufeinander folgenden Masken instinktiv entsteht. Der endliche Automat ist also kein gezielt anvisiertes Ergebnis. So instinktiv wie diese Betrachtungsweise scheint, ist sie in der Literatur bis auf einige Ausnahmen wie Denert in [Denert 1992] nicht beschrieben. Denert beschreibt in [Denert 1992] die Modellierung von Benutzungsschnittstellen in Form von Interaktionsdiagrammen, diese sind eine Art endlicher Automaten.

Eine weitere Möglichkeit, endliche Automaten darzustellen, sind Zustandsdiagramme. Die Dialogabläufe lassen sich, wie bereits im Modell beschrieben, auf einen endlichen Automaten zurückführen. Endliche Automaten können durch Zustandsdiagramme beschrieben werden. Somit können die Dialogabläufe einer Benutzungsschnittstelle mit Hilfe von Zustandsdiagrammen modelliert werden. Wenn also Masken Zustände und die Pfeile zwischen den Masken die Zustandsübergänge abbilden, lässt sich das Modell in Abb. 1 wie folgt in die Form eines Zustandsdiagramms überführen:

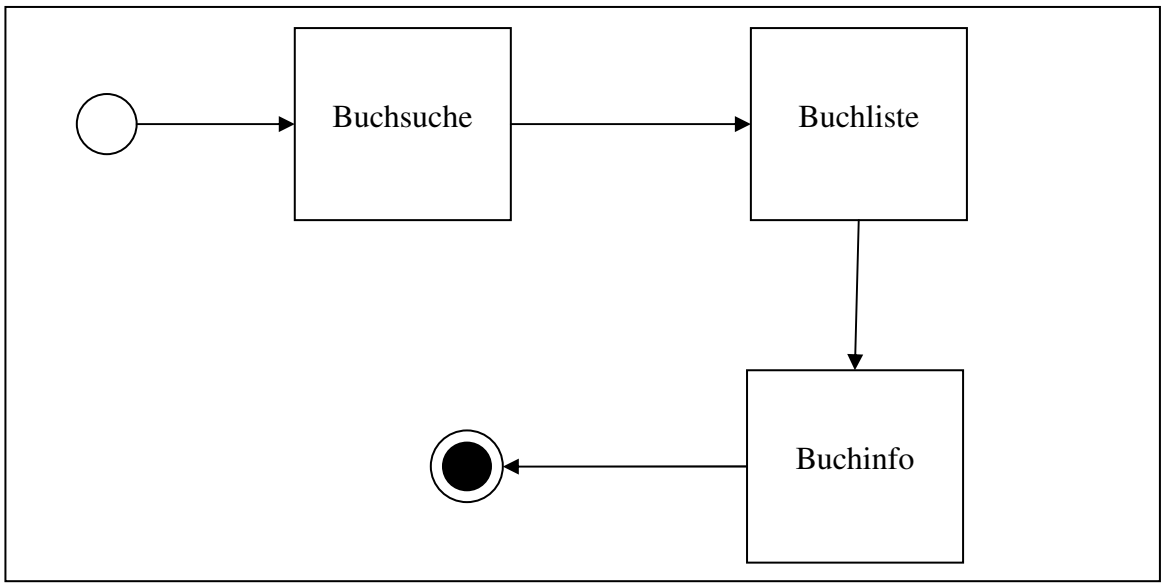


Abb. 2 Zustandsdiagramm Literatur Suche

Das Zustandsdiagramm in Abb. 2 beschreibt zwar die Reihenfolge der Dialogabläufe, wie sie im Modell in Abb. 1 enthalten sind, aber nicht den kompletten Sachverhalt, der dort dargestellt ist. Die Information, unter welchen Bedingungen ein Dialogwechsel stattfindet, ist nicht in Abb. 2 beschrieben. Denn aus Abb. 1 ist ersichtlich, dass beim Klicken auf dem Button „Suche“ ein Dialogwechsel zu der Folgemaske Buchliste erfolgt und beim Klicken auf den Button Details die Maske Buchinfo angezeigt werden. Gerade diese Information fehlt jedoch im Zustandsdiagramm in Abb. 2. Um sie soll es nun angereichert werden.

Dies ist möglich: denn in einem Zustandsdiagramm können Ereignisse definiert werden, die einen Zustandswechsel hervorrufen. Das Zustandsdiagramm von Abb. 2 soll also um die Informationen der Ereignisse ergänzt werden: dies wird in Abb. 3 dargestellt.

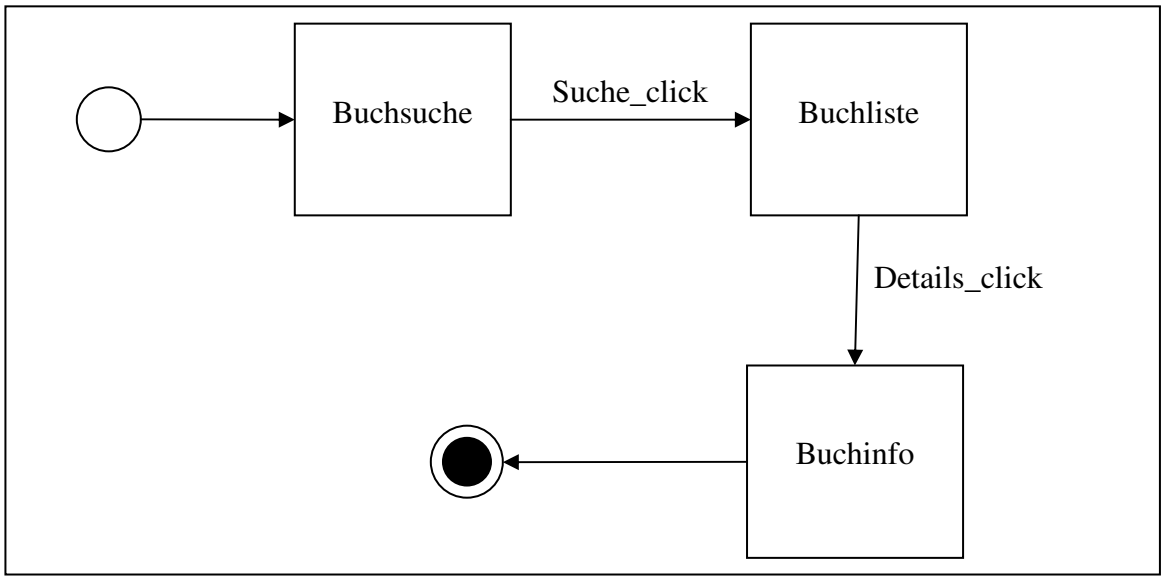


Abb. 3 Zustandsdiagramm Literatur Suche um die Ereignisse angereichert

Vergleicht man das Modell aus Abb. 1 und das nunmehr entstandene Zustandsdiagramm (Abb. 3), erkennt man, dass dieses immer noch nicht alle Details abbildet. Was fehlt, sind die Überlegungen, die sich skizzenhaft in Form von Masken mit GUI-Komponenten wiederfinden. Was z.B. aus dem Zustandsdiagramm nicht hervorgeht, ist die Information, dass durch das Betätigen des Buttons „Suche“ das Ereignis suche_click ausgelöst wird. Das bedeutet, dass die Beziehung zwischen der GUI-Komponente und dem durch den Dialogwechsel auszulösenden Ereignis fehlt. Im Modell aus Abb. 1 ist sie jedoch nachvollziehbar.

Ziel ist es also, das Zustandsdiagramm um Maskenskizzen zu erweitern. Dazu ist es erforderlich, jede Skizze einem Zustand zuzuordnen. Siehe dazu Abb. 4.

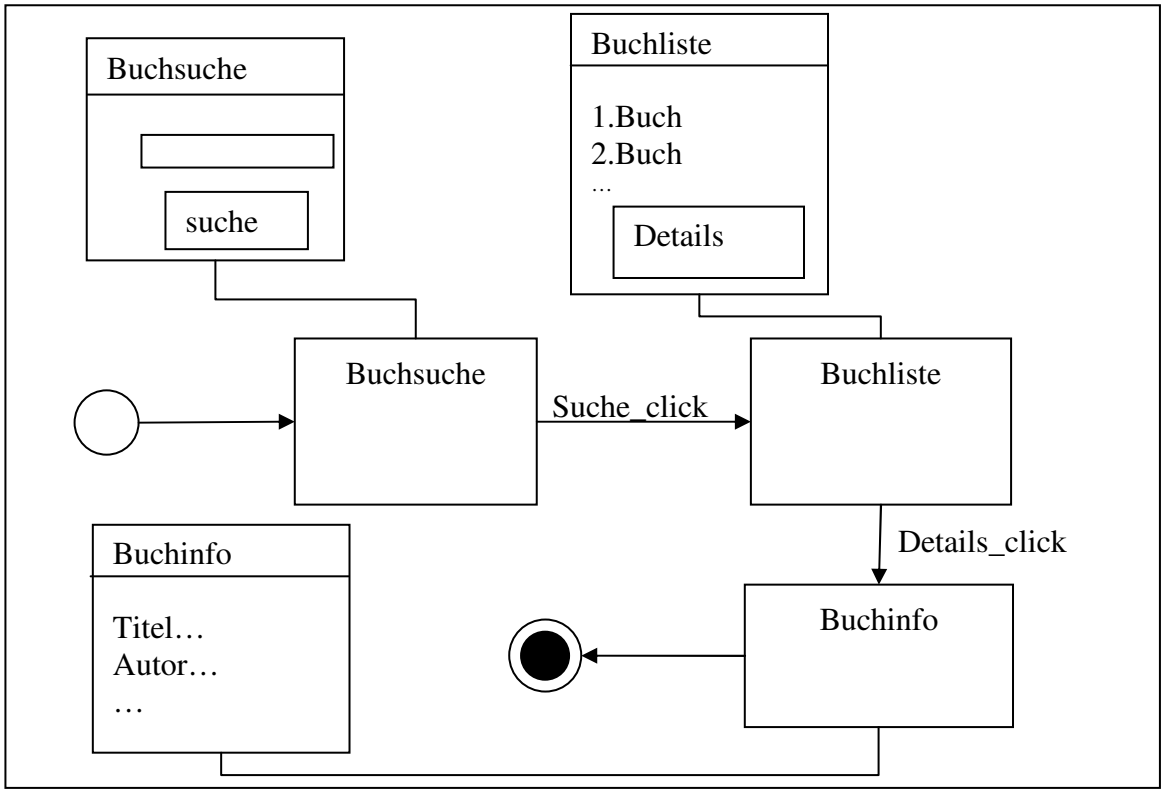


Abb. 4 Zustandsdiagramm Literatur Suche um die Maskenskizzen angereichert

Die gesamten Informationen aus dem Ursprungsmodell in Abb. 1 wurden in Form eines Zustandsdiagramms und Fensterskizzen beschrieben. Dialogabläufe der Buchsuche wurden in Form eines Zustandsdiagramms modelliert. Hierzu wurden Zustände, Zustandsübergänge und Ereignisse definiert und die Zustände Skizzen zugeordnet.

In Abb. 1 und im Modell in Abb. 4 handelt es sich letztendlich um dasselbe Modell. Das Modell aus Abb. 1 ist durch den Versuch, die Dialogabläufe in ihrer Reihenfolge darzustellen, entstanden. Bei diesem Vorgang ist intuitiv, also eher unbeabsichtigt, ein Zustandsdiagramm entstanden. Das Modell in Abb. 4 bildet auch den Versuch, die Dialogabläufe in ihrer Reihenfolge darzustellen, mit dem Unterschied, dass hier das Modell gezielt auf ein Zustandsdiagramm hin entwickelt worden ist.

Wenn es sich in Abb. 1 und Abb. 4 um ein und dasselbe Modell handelt, kann man doch die Dialogabläufe gleich in Form eines Zustandsdiagramms, wie sie in Abb. 4 dargestellt sind, modellieren, da Zustandsdiagramme viele Konstrukte bieten, die für die Modellierung einer Benutzungsschnittstelle sinnvoll eingesetzt werden können.

Da es ohnehin nötig ist, die Masken zu skizzieren, ist es doch zweckmäßig, diese Skizzen mit Hilfe eines GUI-Builders zu erzeugen. Das Ergebnis ist dann ein ausführbarer Code der Maske.

In einem Zustandsdiagramm können in jedem Zustand Aktivitäten definiert werden. Dabei handelt es sich um Operationen, die ausgeführt werden, wenn sich das Zustandsdiagramm in dem Zustand befindet. Wurden nun ausführbare Skizzen mit einem GUI-Builder generiert, können diese als Aktivität in einem Zustand definiert werden. Auf diese Weise wird beim Eintreten eines Zustands die dazugehörige GUI ausgeführt.
Im Folgenden (Abb. 5) ist das Zustandsdiagramm aus Abb. 4 dargestellt, das um den ausführbaren Code der Masken in Form von Aktivitäten angereichert wurde:

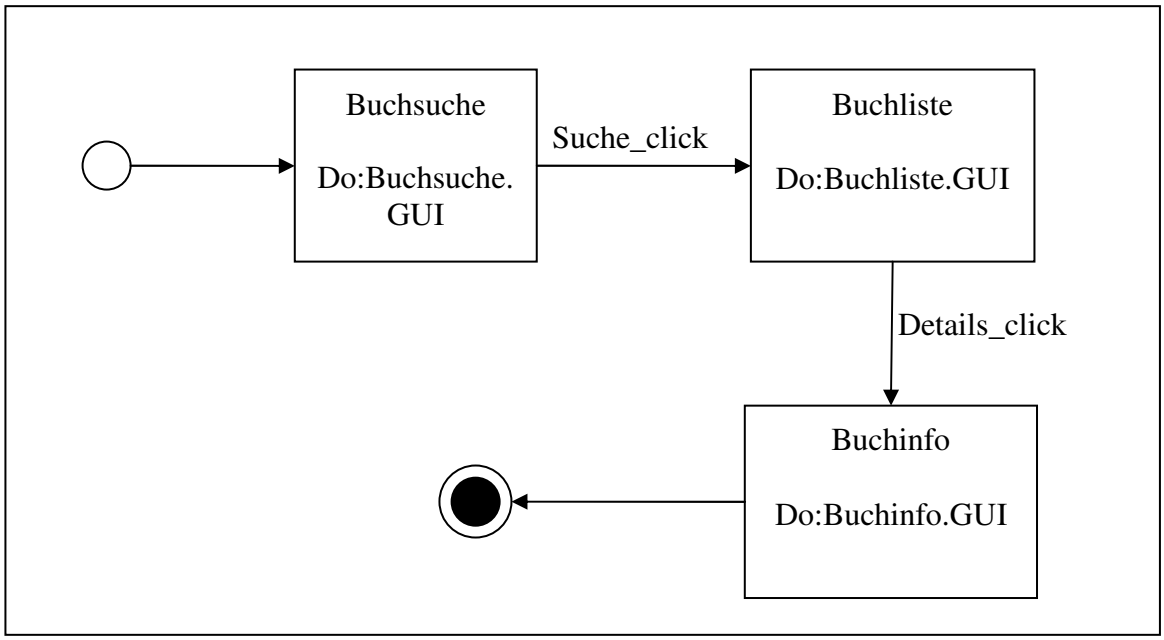


Abb. 5 Zustandsdiagramm Literatur Suche um die Masken angereichert

In Abb. 5 wurden Masken mit Hilfenahme eines GUI-Builders generiert und diese den Zuständen im Zustandsdiagramm zugeordnet. Buchsuche.GUI, Buchliste.GUI und Buchinfo.GUI sind Verweise auf den vom GUI-Builder generierten Maskencode. Dieser Code wird automatisch beim Betreten eines Zustands ausgeführt.

Am Anfang befindet sich das Zustandsdiagramm im Zustand Buchsuche und die Operation Buchsuche.GUI wird ausgeführt, sodass die Maske Buchsuche erscheint. Durch einen Klick auf den Button Suche, wird das Ereignis Suche_click ausgelöst. Das Zustandsdiagramm wechselt zum Folgezustand Buchliste. Beim Betreten des Zustands Buchliste wird die Aktivität Do:Buchliste.GUI ausgeführt und es wird die GUI-Maske der Buchliste angezeigt. Von hier aus kann durch das Auslösen von Details_click ein Zustandsübergang angestoßen werden und das Zustandsdiagramm wechselt in den Zustand Buchinfo. In Buchinfo wird die Maske Buchinfo auf dem Bildschirm dargestellt.

Werden die Dialogabläufe einer Benutzungsschnittstelle in einem formalen eindeutigen Modell beschrieben und das Modell um die Bildschirmmasken angereichert, so ist wenig zusätzlicher Aufwand erforderlich, um das Modell automatisch simulieren zu können, und die Dialogabläufe gemäß dem Modell ablaufen zu lassen.

In einem Zustandsdiagramm können außerdem noch Aktionen definiert werden, die ausgeführt werden, wenn ein Ereignis einen Zustandsübergang auslöst. Im Beispiel der Literatur-Suche z.B. löst ein Klick auf den Button suche ein Ereignis aus. Dieses Ereignis stößt eine Aktion an. Die angestoßene Aktion führt die Suche aus. Es bietet sich also an, das Zustandsdiagramm um Informationen der auszuführenden Aktionen anzureichern.

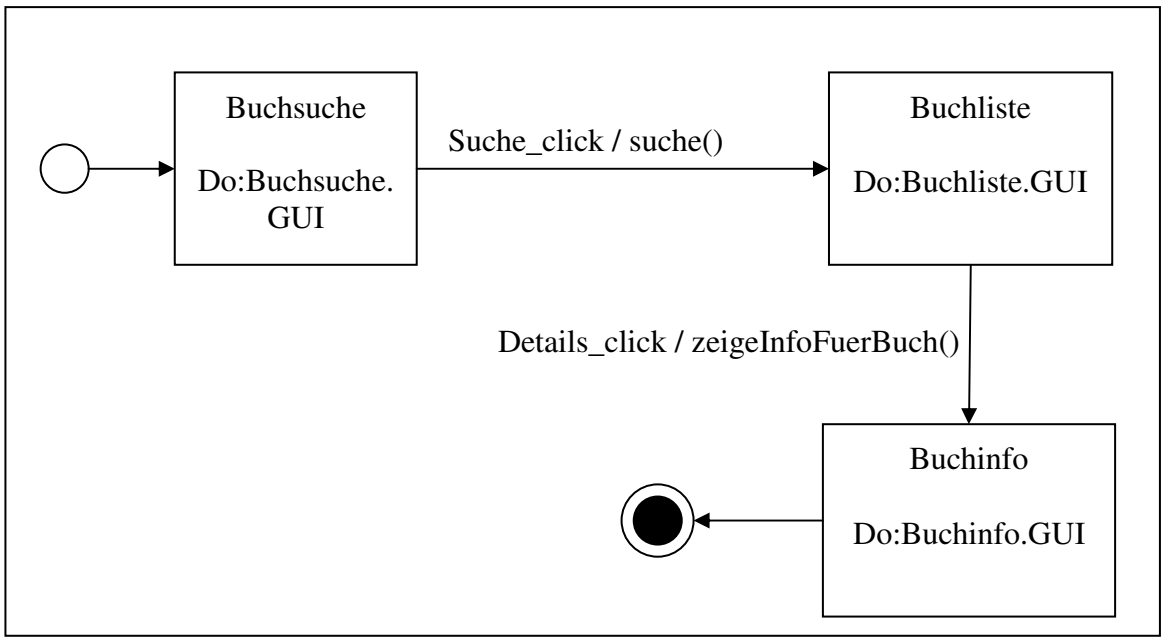


Abb. 6 Zustandsdiagramm Literatur Suche um die Aktionen angereichert

Wie in Abb. 6 dargestellt wurde das Zustandsdiagramm aus Abb. 5 um die Aktionen ergänzt. Suche_click z.B. löst die Aktion suche() aus.

Ist das dynamische Verhalten einer Benutzungsschnittstelle erstmal in der in Abb. 6 beschriebenen Form, eröffnet es uns eine neue Möglichkeit, die der automatischen Codegenerierung. Informationen an einem Zustandsdiagramm können dazu benutzt werden,

um automatisch Code zu generieren. Die im Zustandsdiagramm aus Abb. 6 definierten Aktionen, suche() und zeigeInfoFuerBuch(), lassen sich automatisch genieren und die dahinter liegende Funktionalität kann dann später implementiert werden. Es kann also eine Klasse automatisch erzeugt werden, die alle Methodendeklarationen enthält, eine so genannte Fassaden-Klasse. Die Fassaden-Klasse ist eine Schnittstelle zwischen der Interaktionskomponente und der Funktionskomponente. Sie soll dazu dienen, Implementierungsdetails zu verbergen. So ruft die Interaktionskomponente Funktionen der Fassaden-Klasse auf. Diese wiederum ruft Funktionen der Funktionskomponente auf (Abb. 7).

Diese Methode bietet darüber hinaus den Vorteil der Sprachunabhängigkeit. Werden Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen beschrieben, kann das entstandene Modell in einer beliebigen Programmiersprache umgesetzt werden. Im Rahmen dieser Bachelor-Arbeit beschränken wir uns jedoch auf eine bestimmte Programmiersprache.

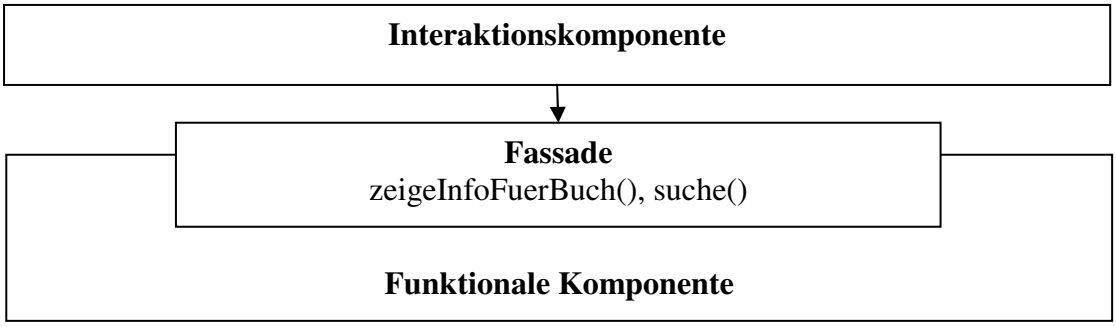


Abb. 7 Fassade

Zustandsdiagramme besitzen also genügend Ausdruckskraft, um das dynamische Verhalten von Benutzungsschnittstellen vollständig zu spezifizieren. In einem Zustandsdiagramm stehen die Interaktionen des Benutzers mit dem System, ihre Reihenfolge und die von ihm wahrnehmbaren Zustandsänderungen im Vordergrund. Eine GUI befindet sich immer für einen gewissen Zeitraum in einem bestimmten Zustand. Ein Zustand markiert eine Stelle im Dialogablauf, an der das System auf eine Eingabe des Benutzers wartet. Benutzereingaben wie z.B. das Drücken eines Buttons mit der Maus lösen Ereignisse aus. Ereignisse stoßen Aktionen an. Nach Ausführung der Aktion gelangt die Benutzungsschnittstelle in den Folgezustand.

Die Methode Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen zu modellieren und das entstandene Zustandsdiagramm zu simulieren und aus dem Modell Code in Form einer Fassade automatisch zu generieren, ist jedoch in der verfügbaren Literatur nicht beschrieben. Daher liegt die Vermutung nahe, dass noch kein Tool existiert, das diese Methode umsetzt. Unsere Recherche nach solch einem Tool bestätigte unsere Vermutung (siehe Kapitel 2). Daher ist Ziel dieser Bachelor-Arbeit, solch ein Tool zu realisieren.

1.2.2 Vision SUIT

SUIT steht für Statechart User-Interface Tool. Es handelt sich um das Tool, das im Rahmen dieser Bachelor-Arbeit entwickelt werden soll. SUIT soll die in der Vision in Abschnitt 1.2.1

beschriebene Methode umsetzen. Es unterstützt also die Modellierung von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen.

SUIT ermöglicht die Modellierung des dynamischen Verhaltens einer Benutzungsschnittstelle in Form eines Zustandsdiagramms. Das in SUIT erstellte Zustandsdiagramm kann um die Informationen der Bildschirmmasken angereichert werden. In ihm wird festgehalten, welche Maske auf dem Bildschirm in einem Zustand dargestellt werden soll. Auf diese Art und Weise wird die Lücke zwischen der Modellierung des dynamischen Verhaltens einer Benutzungsschnittstelle und die graphische Gestaltung ihrer Oberfläche geschlossen. In SUIT wird außerdem noch erreicht, dass der in die Modellierung in Form von Zustandsdiagrammen und Fensterentwürfen investierte Aufwand in den Entwicklungsprozess mehr integriert wird. SUIT steigert den Nutzen eines Modells, indem aus dem Modell automatisch Code in Form einer Fassade generiert werden kann. Darüber hinaus kann das Modell simuliert werden.

Die Simulation dient dazu, das spezifizierte dynamische Verhalten einer Benutzungsschnittstelle zu überprüfen. Da die dynamische Verhaltensspezifikation durch Zustandsdiagramme erfolgt, muss die Simulation die Zustandsdiagramme in dem Sinne animieren, dass der Ablauf der Zustandsfolgen mit den damit verbundenen Bildschirmmasken graphisch visualisiert wird. Die Simulationsmöglichkeit ermöglicht somit ein exploratives Prototyping. Dieses hilft dabei, die Kommunikation zwischen den Entwicklern und den Benutzern zu erleichtern. Lauffähige Prototypen sind das bestmögliche Kommunikationsmittel über eine Benutzungsschnittstelle. Sie werden in der Anforderungsanalyse angewandt, um die Anforderungen von Benutzern an das künftige System zu identifizieren.

Mit SUIT sollen Prototypen mit möglichst geringem Aufwand und mit so wenig Kosten wie möglich erstellt werden. Außerdem soll es in SUIT die Möglichkeit geben, Prototypen nicht nur als Wegwerfprototypen zu realisieren, sondern diese auch zu einem ausgereiften System weiterentwickeln zu können. Die Kosten und der Aufwand im Rahmen der Realisierung einer Benutzungsschnittstelle soll auch durch die automatische Codegenerierung reduziert werden. Durch die automatische Erzeugung der Fassade entfällt hiermit ein weiterer manueller Programmieraufwand.

1.2.3 SUIT Komponenten

Die zentrale Anforderung an das Tool dieser Bachelor-Arbeit besteht darin, Benutzungsschnittstellen in Form von Zustandsdiagrammen und Bildschirmmasken zu erzeugen. Zu diesem Zweck sind ein Zustandsdiagrammeditor und ein GUI-Builder unabdingbar.

Es existieren sowohl kommerzielle als auch nichtkommerzielle Zustandsdiagrammeditoren und GUI-Builder auf dem Markt (siehe Marktanalyse in Kapitel 2). Daher liegt die Überlegung nahe, jeweils eines dieser Produkte, sofern es erweiterbar und integrierbar ist, wieder zu verwenden. In diesem Fall wäre die Realisierung eines eigenen GUI-Builders oder Zustandsdiagrammeditors überflüssig. Die Kommunikation zwischen dem GUI-Builder und dem Zustandsdiagrammeditor muss durch eine eigene Komponente realisiert werden. Der Vorteil dieser Lösung liegt in der Möglichkeit der Wiederverwendung bereits bestehender Komponenten.

Trotzdem haben wir uns zur Umsetzung des Tools dieser Bachelor-Arbeit dafür entschieden, einen eigenen Zustandsdiagrammeditor zu realisieren, da es unser Ziel ist, einen speziell für Benutzungsschnittstellen maßgeschneiderten Zustandsdiagrammeditor zu entwickeln.

Im Rahmen dieser Bachelor-Arbeit soll ein explorativer Prototyp erstellt werden. Es soll überprüft werden, ob die Idee, Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen zu entwickeln, praxistauglich ist. Stellt sich anschließend heraus, dass Zustandsdiagramme in der jetzigen Notation für die Modellierung des dynamischen Verhaltens von Benutzungsschnittstellen unzureichend sind, so kann man mit einer eigenen Realisierung flexibler auf neue Erkenntnisse reagieren.

Erfordern z.B. neue Erkenntnisse eine Abänderung der Notation des Zustandsdiagramms, so kann diese Änderung problemlos vorgenommen werden. Eine eigene Implementierung bietet mehr Flexibilität, um auf Änderungen der Anforderungen und auf neue Erkenntnisse zu reagieren. Daher besteht unser Vorgehen darin, für das Tool dieser Bachelor-Arbeit einen existierenden GUI-Builder wiederzuverwenden und einen eigenen Zustandsdiagrammeditor zu implementieren.

1.2.4 Einsatz SUIT

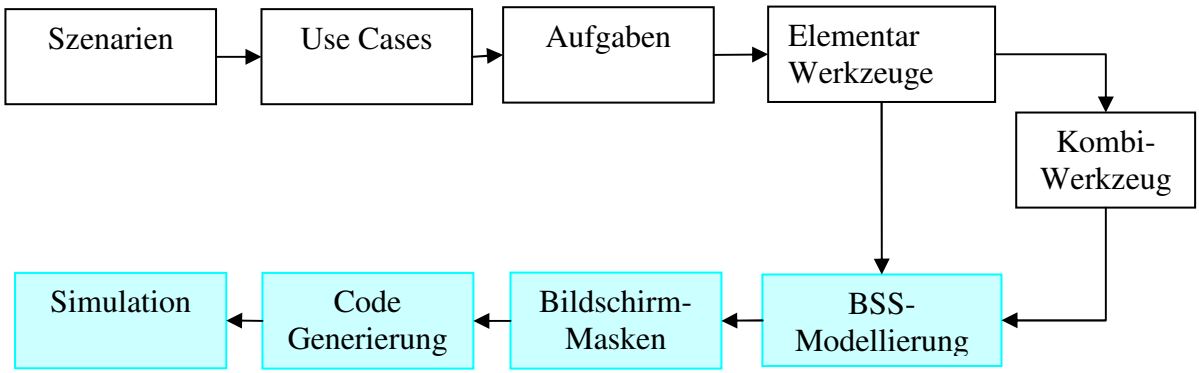


Abb. 8 Einsatz von SUIT

Abb. 8 zeigt die Etappen an, in denen das Tool dieser Bachelor-Arbeit zum Einsatz kommen kann. Dieser Einsatz erfolgt in der Regel, nachdem die im WAM-Ansatz [Züllighoven 1998] beschriebenen Werkzeuge identifiziert worden sind.

Arbeitsabläufe lassen sich in Szenarien beschreiben. Mit Hilfe von Szenarien werden Use Cases identifiziert. Anhand der Anwendungsfälle und Szenarien werden wiederkehrende Aufgaben erkannt und zu Elementar-Werkzeugen zusammengefasst. Bei komplexen Aufgaben können elementare Werkzeuge zu einem Kombi-Werkzeug kombiniert werden. Diese Zusammenführung von Subwerkzeugen zu einem Kombi-Werkzeug wird im WAM-Entwurfsmuster Werkzeugkomposition beschrieben [Züllighoven 1998]. An diesem Punkt kommt das Tool dieser Bachelor-Arbeit zum Einsatz. SUIT schreibt jedoch keine bestimmte Reihenfolge der Arbeitsabfolgen vor. Abb. 8 beschreibt die gängige Vorgehensweise. Wichtig ist, dass die Anforderungen an eine Benutzungsschnittstelle erkannt worden sind, um daraus ein Modell ihres dynamischen Verhalten in SUIT modellieren und aus diesem einen Prototyp generieren zu können. Ein detailliertes Verständnis der Anforderungen ist jedoch keine Vorraussetzung, da SUIT aus einem Modell einen Prototyp erstellt, um diesen dem Anwender frühzeitig vorzuführen und mit ihm die Anforderungen der Benutzungsschnittstelle abzustimmen.

In diesem Tool werden Dialogabläufe der bereits identifizierten Werkzeuge in Form von Zustandsdiagrammen modelliert und den Zuständen Bildschirmmasken zugeordnet. Das Tool generiert aus dem entstandenen Modell Code. Außerdem kann das Modell simuliert werden.

Der praktische Einsatz von SUIT wird in der Fallstudie (siehe Kapitel 7) detailliert beschrieben.

1.3 Aufbau der Arbeit

Im zweiten Kapitel wird die Notwendigkeit der Realisierung von SUIT begründet. Im ersten Schritt wird eine Marktanalyse unternommen, die feststellen soll, ob bereits Tools für die Erzeugung von Benutzungsschnittstellen und Fensterentwürfen auf dem Markt existieren. Da diese Untersuchung zu keinem positiven Ergebnis kommt, werden im zweiten Schritt auf dem Markt existierende Komponenten von SUIT (GUI-Builder und Zustandsdiagramm-Editor) analysiert.

Die nächsten drei Kapitel dienen der theoretischen Grundlegung der Entwicklung von SUIT. In Kapitel 3 wird die Theorie der Erzeugung von Benutzungsschnittstellen in Form von endlichen Automaten und Fensterentwürfen dargelegt. In Abgrenzung von der Theorie Denerts, der Erzeugung von Benutzungsschnittstellen mit Interaktionsdiagrammen und Fensterentwürfen, wird die im Rahmen dieser Bachelor-Arbeit geplante Vorgehensweise der Erzeugung von Benutzungsschnittstellen auf Basis von Zustandsdiagrammen und Fensterentwürfen beschrieben. In Kapitel 4 werden alle Anforderungen an SUIT dargelegt und klassifiziert, in Kapitel 5 die SUIT-Architektur beschrieben.

Im dritten Bereich der Arbeit wird die Umsetzung von SUIT dargestellt und nach verschiedenen Gesichtspunkten untersucht und beurteilt.

In Kapitel 6 wird ein kleiner Einblick in die Umsetzung von SUIT gegeben. Außerdem wird auf Besonderheiten bei der Realisierung von SUIT und auf die im Rahmen der Umsetzung verwendeten Technologien eingegangen.

In Kapitel 7 wird am Beispiel einer Fallstudie der praktische Einsatz von SUIT demonstriert. In Kapitel 8 folgt eine Analyse des Usabiliy-Tests, dem SUIT mit Hilfe von Testpersonen unterzogen wurde. Hierdurch soll SUIT auf seine Praxistauglichkeit hin untersucht werden.

Der vierte Teil der Arbeit besteht aus den Kapiteln 9 und 10. In Kapitel 9 werden die Möglichkeiten und Grenzen von SUIT beschrieben. Und in Kapitel 10 folgt ein Ausblick darüber wie sich SUIT weiterentwickeln lässt. Hier werden für SUIT sinnvolle Erweiterungsvorschläge gemacht und diskutiert.

In Kapitel 11 folgt das Resumé.

Folgende Aufteilung fand im Rahmen der Ausarbeitung statt:
Kapitel 2 .4 und 7 wurden von Noura Haji bearbeitet.
Kapitel 2.5, 9 und 10 wurden von Vinh-Phu Phan bearbeitet.
Die restlichen Kapiteln 2.1, 2.2, 2.3, 3, 4, 5, 6, 8 und 11 sind durch gemeinsame Arbeit entstanden.

2. Marktanalyse

2.1 Einleitung

Ein Ziel dieser Marktanalyse ist es, zu überprüfen, ob bereits Tools für die Erzeugung von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen auf dem Markt existieren.

Als Leitfaden für die Untersuchung diente eine grobe Anforderungsliste an SUIT, die im Abschnitt 2.2 angegeben wird. Das Ergebnis dieser Untersuchung folgt in Abschnitt 2.3.

Der im Rahmen unserer Bachelor-Arbeit zu entwickelnde Prototyp SUIT hat als Anforderung, grafische Oberflächen anhand von Zustandsdiagrammen und Fensterentwürfen zu erzeugen. Unabdingbar für ihn sind daher ein Zustandsdiagramm-Editor und ein integrierbarer GUI-Builder (Siehe 1.2.3).

Im Abschnitt 2.4 werden vorhandene GUI-Builder zur Erzeugung graphischer Benutzeroberflächen betrachtet. Diese Analyse hat das Hauptziel, einen geeigneten GUI-Builder zwecks Wiederverwendung für das Tool dieser Bachelor-Arbeit zu finden.

Im Abschnitt 2.5 werden vorhandene Zustandsdiagramm-Editoren untersucht. Das Hauptaugenmerk liegt hierbei auf der Suche nach neuen Anforderungen, da im Rahmen dieser Bachelor-Arbeit die Realisierung eines eigenen Zustandsdiagramms-Editor geplant ist.

Im Abschnitt 2.6 folgt ein Fazit.

2.2 Grobe Anforderungen

SUIT erlaubt die Modellierung des dynamischen Verhaltens von Benutzungsschnittstellen in Form von Zustandsdiagrammen. Diese können außerdem um weitere Details der graphischen Oberfläche bereichert werden, wobei für jeden Zustand festgelegt wird, welche Masken auf dem Bildschirm dargestellt werden. Das erstellte Zustandsdiagramm kann simuliert werden. Außerdem wird eine Fassade für die Weiterentwicklung generiert (siehe 1.2.2). Somit bestehen im Groben zwei Anforderungen an SUIT: Benötigt werden Zustandsdiagramm-Editor, der über die gängigen Operationen zum Anlegen eines Zustandsdiagramms verfügt, und ein GUI-Builder, der in den Zustandsdiagramm-Editor integriert ist, so dass für jeden Zustand innerhalb des Zustandsdiagramms per Drag & Drop Masken erzeugt werden können (siehe 1.2.3).

In Tabelle 1 sind alle notwendigen Anforderungen an SUIT aufgelistet.

Anforderung	Beschreibung
Erzeugen eines Zustands	Ein Zustand kann auf der Zeichenfläche platziert werden
Erzeugen eines Anfangszustandes	Ein Anfangzustand kann auf der Zeichenfläche platziert werden.
Erzeugen eines Zustandsübergangs (Transition)	Ein Zustandsübergang kann auf der Zeichenfläche platziert werden.
Neuanlegen von Masken	Masken können erzeugt werden.
Erzeugung von Masken zu den Zuständen	Jedem Zustand kann eine Maske zugeordnet werden.
Speicherung von Zustanddiagrammen	Das Zustandsdiagramm muss gespeichert werden können.
Simulation	Das Zustandsdiagramm kann simuliert werden.
Fassade	Es wird eine Fassade generiert werden.
GUI-Masken-Ereignisse den Zustandsübergang zuweisen	Es muss die Möglichkeit geben, Ereignisse der GUI-Maske den Zustandsübergang zuzuweisen

Tabelle 1 Notwendige Anforderungen an SUIT

2.3 Existierende Tools

Im Rahmen dieses Kapitels wurde untersucht, ob es auf dem Markt Tools für die Erzeugung von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen existieren. Als Leitfaden für die Untersuchung diente die grobe Anforderungstabelle aus 2.2. Die Recherche nach bereits existierenden, kommerziellen oder nicht kommerziellen, Softwarelösungen ergab jedoch kein Ergebnis für eine der Vision entsprechende Gesamtlösung.

2.4 GUI-Builder

Als Entwicklungsumgebung für die Entwicklung von SUIT kommen insbesondere Eclipse [eclipse] und Microsoft Visual Studio.Net [Visual Studio.Net 2003] in Betracht. Microsoft Visual Studio verfügt über einen fest eingebauten GUI-Builder. Eclipse kann mit einem beliebigen GUI-Builder-Plugin ausgestattet werden. Die drei bekanntesten Produkte sind Jigloo [Jigloo 3.9.5], Window Builder Pro [WindowBuilderPro] und Visual Editor [VisualEditor]. Bei der Analyse haben wir uns für Eclipse auf Jigloo beschränkt.

Die Analyse der GUI-Builder ist wesentlich, da sie die Weichen für das weitere Vorgehen stellt. Denn mit der Entscheidung für einen bestimmten GUI-Builder fällt die Entscheidung für eine bestimmte Entwicklungsumgebung für SUIT.

2.4.1 Jigloo

CloudGardens Jigloo GUI-Builder [Jigloo 3.9.5] ist ein Add-In für Eclipse. Eclipse ist ein Open-Source-Projekt. Die Entwicklungsumgebung ist frei verfügbar und gut für die Entwicklung von Javaprogrammen geeignet. Eclipse wird nicht installiert, sondern kann als Java-Anwendung sofort ausgeführt werden.

Oberfläche:

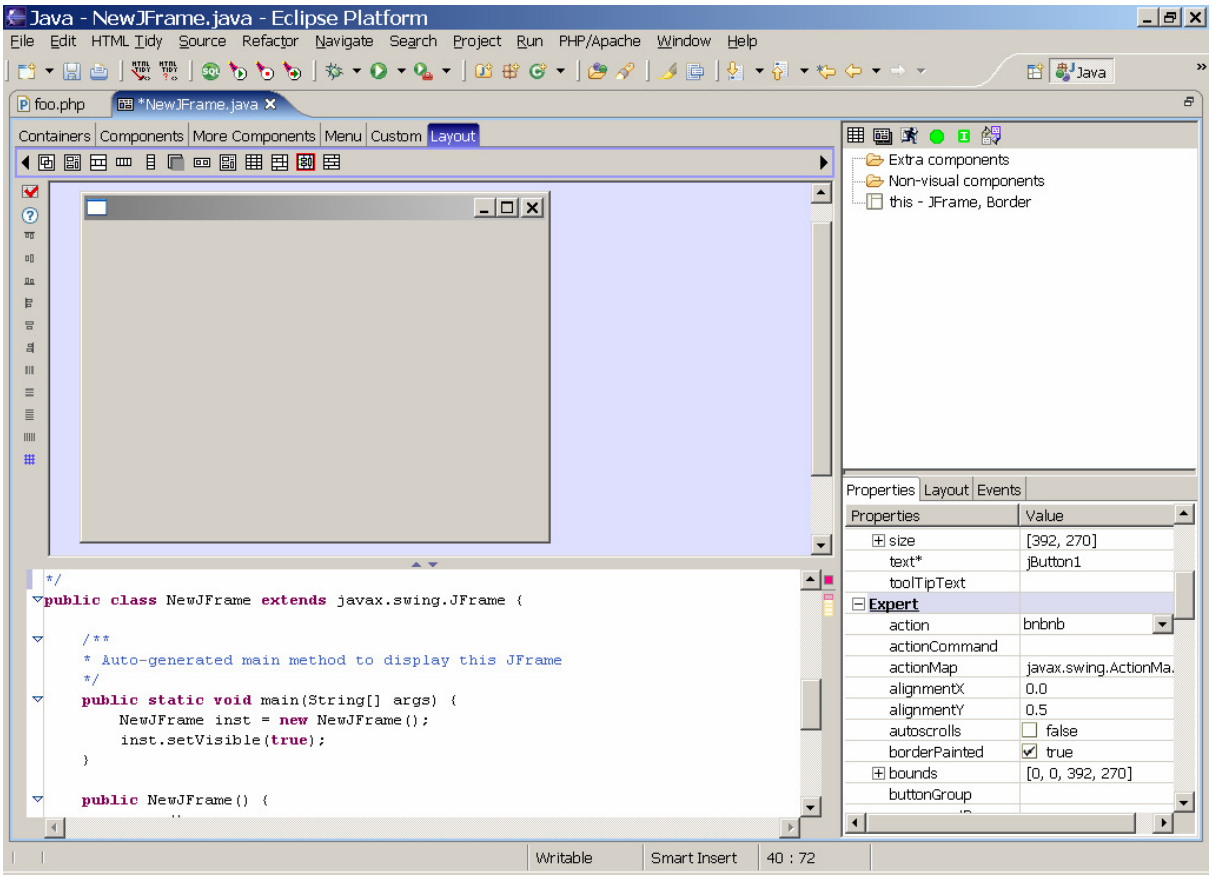


Abb. 9 Jigloo in Eclipse

#

Abb. 9 zeigt die wichtigsten Komponenten von Jigloo in Eclipse in der vollen Ansicht:

- Form Editor (GUI-Editor)
- Source Editor (Quelltext-Editor)
- Outline View (oben rechts)
- Property Editor (unten rechts)

Bei der standardmäßigen Einstellung erscheinen Quelltext und GUI - wie in Abb. 9 zu sehen - auf derselben Oberfläche. So können durch die gleichzeitige Darstellung von Entwurfsmodus und Quelltext Änderungen im Quelltext auch während der GUI-Erzeugung eingesehen werden.

Die Outline View bietet eine gute Übersicht über alle bereits erzeugten GUI-Komponenten. Die verfügbaren GUI-Komponenten sind in der Palette thematisch geordnet (siehe Abb. 10), was ihre Bedienung äußert erleichtert.

Abb. 10 Die Palette in Jigloo

Auch der Property Editor (Abb. 11) ist anschaulich und wohlgeordnet. Er besteht aus drei verschiedenen Fenstern:

- Properties
- Layout
- Events

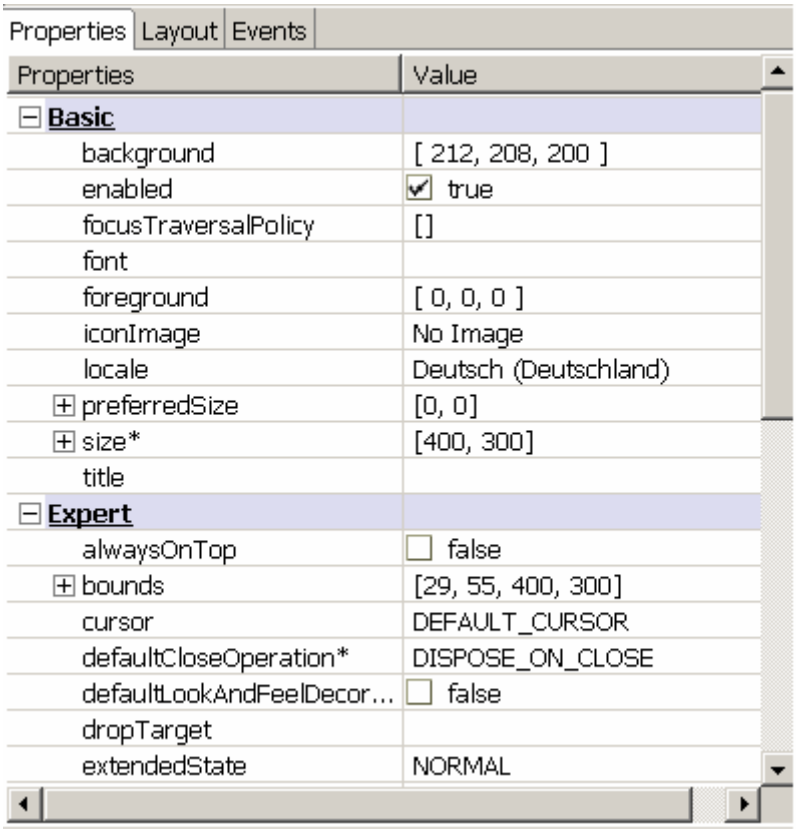


Abb. 11 Property Fenster in Jigloo

Bedienung:

Jigloo bietet einen Layout Manager. Standardmäßig ist ein Border-Layout für das Entwurfsfenster ausgewählt. Für Entwickler, die das Arbeiten mit dem Layout Manager nicht gewohnt sind, ist es oft irritierend, wenn deshalb die eingefügten Komponenten nicht an der gewünschten Stelle erscheinen. Für Anfänger, die mit dem Konzept der Layout Manager nicht vertraut sind, kann es viel Mühe und Zeit kosten, die Oberfläche bedarfsgerecht zu gestalten. Grundsätzlich ist es aber auch möglich, Komponenten ohne einen Layout-Manager in einem Container zu platzieren. In diesem Fall sollte das absolute Layout ausgewählt werden. Die

Komponenten können mit einem Klick ausgewählt und mit einem weiteren Klick an der damit zugleich selektierten Stelle der Zeichenfläche eingefügt werden.

Jigloo unterstützt nicht das Einfügen von GUI-Komponenten über Drag&Drop. Außerdem kann die Größe einer GUI-Komponente nicht gleich beim Erzeugen geändert werden. Sie besitzt immer eine vordefinierte Default-Größe. Erst nach dem Platzieren auf dem Form-Editor kann sie angepasst werden.

Beim Erzeugen von GUI-Komponenten wird ein Fenster zum Bestätigen oder Ändern von Default-Werten angezeigt, welches man mit OK bestätigen muss. Auch beim Löschen von GUI-Komponenten werden Bestätigungsfenster angezeigt, die – im Falle einer Löschung – mit OK bestätigt werden müssen. Wenn mehrere GUI-Komponenten erzeugt werden müssen; wirkt diese ständige Abfrage störend und macht die Bedienung unkomfortabel.

Codegenerierung:

Durch die Formatierung mit Hilfe von Tabulatoren lässt sich der erzeugte Java-Code übersichtlich und anschaulich darstellen.

```
private void initGUI() {
    try {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        getContentPane().setLayout(null);
        {
            jButton1 = new JButton();
            getContentPane().add(jButton1);
            jButton1.setText("jButton1");
            jButton1.setBounds(28, 21, 140, 28);
        }
        {
            jButton2 = new JButton();
            getContentPane().add(jButton2);
            jButton2.setText("jButton2");
            jButton2.setBounds(217, 21, 140, 28);
        }
        pack();
        setSize(400, 300);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

A

B

Abb. 12 Von Jigloo generierter Code-Ausschnitt einer GUI mit 2 JButtons

Abb. 12 stellt einen generierten Code-Ausschnitt von Jigloo dar. Die dazugehörige Maske besteht aus zwei JButtons, JButton1 (A) und JButton2 (B). In Jigloo werden alle zu einer erzeugten GUI-Komponente gehörigen Codestellen in geschweifte Klammern gesetzt. Wird z.B. der JButton1 entfernt, so wird der dazugehörige Code (Block A) gelöscht. Manuell generierter Code innerhalb dieser Klammernblöcke gehen beim Löschen der betreffenden GUI-Komponente daher verloren.

```
private void initGUI() {
    try {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        getContentPane().setLayout(null);
        {
            jButton1 = new JButton();
            getContentPane().add(jButton1);
            jButton1.setText("jButton1");
            jButton1.setBounds(28, 21, 140, 28);
            jButton1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent evt) {
                    System.out.println("jButton1.actionPerformed, event="
                        + evt);
                    //TODO add your code for jButton1.actionPerformed
                }
            });
        }
        {
            jButton2 = new JButton();
            getContentPane().add(jButton2);
            jButton2.setText("jButton2");
            jButton2.setBounds(217, 21, 140, 28);
        }
        pack();
        setSize(400, 300);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

A

Abb. 13 Code-Ausschnitt einer GUI mit 2 JButtons und ActionListener in Jigloo

Fügt man einer GUI-Komponente (in Abb. 13 JButton1) einen ActionListener vom Typ Anonymous hinzu, so wird der Code wie Block A in Abb.8 dargestellt. Wird nun in der Methode actionPerformed() Code einprogrammiert, wird dieser beim Löschen des ActionListeners über Jigloo ebenfalls gelöscht. Vorteilhaft ist hierbei jedoch, dass beim erneuten Erzeugen eines ActionListeners des Typs Anonymous über Jigloo der vom Anwender erzeugte Code wieder eingefügt wird, sofern noch keine Speicherung des Codes erfolgt ist.

2.4.2 Integrierter GUI-Builder von Microsoft Visual Studio.Net

Der GUI-Builder ist ein fester Bestandteil von Visual Studio.Net [Visual Studio.Net 2003]. Microsoft Visual Studio.Net ist nicht frei erhältlich. Die Entwicklungsumgebung unterstützt die Entwicklung von Anwendungen für die .Net Plattform. Es werden unter anderem die Programmiersprachen Visual Basic, C#, J# und C++ unterstützt.

Oberfläche:

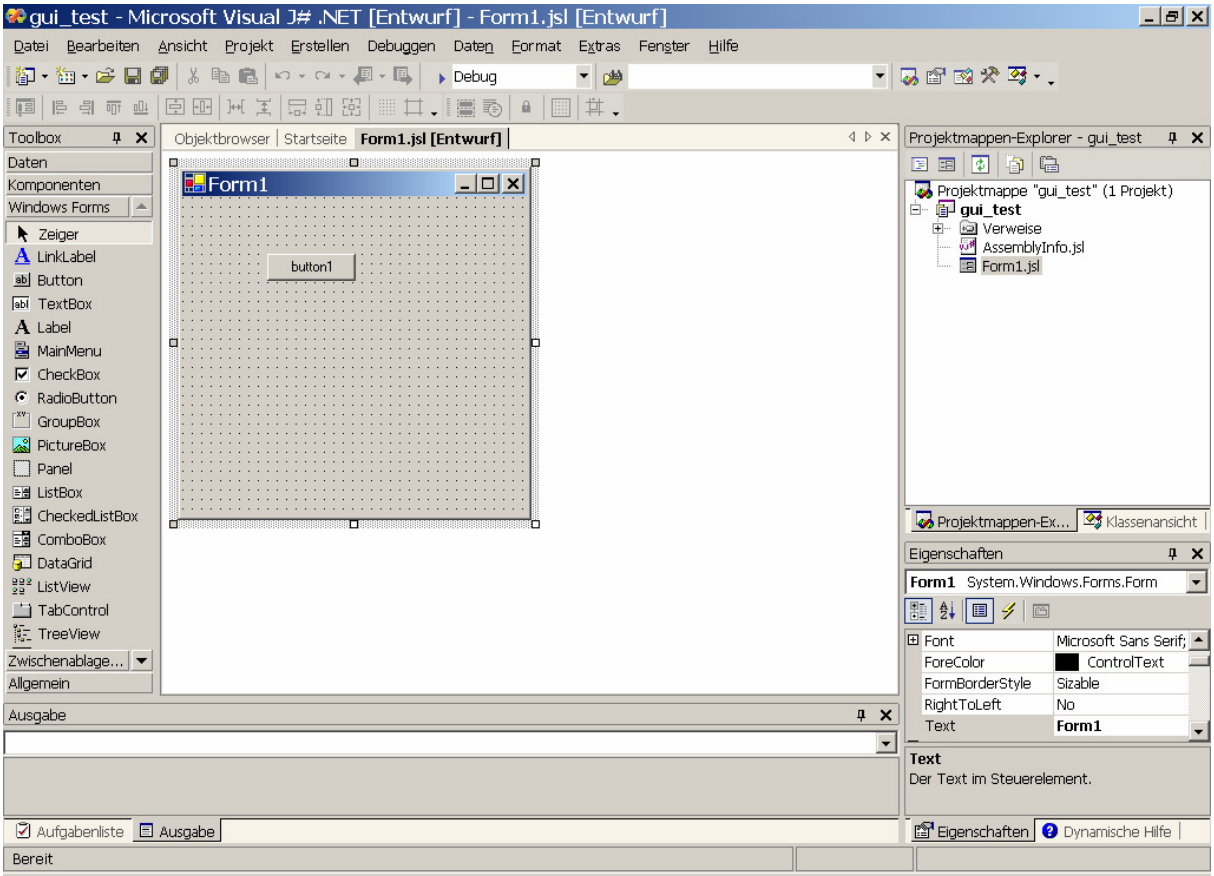


Abb. 14 integrierter GUI-Builder von Microsoft Visual Studio.Net 2003

Die Programmoberfläche von Microsoft Visual Studio in der GUI-Builder Standard-Ansicht besteht aus vier Kernkomponenten (Abb. 14)

- Toolbox: Die Palette nennt sich in Visual Studio Toolbox. Hier sind die wichtigsten GUI-Komponenten verankert. Die Toolbox ist mit Komponenten überfüllt und wirkt dadurch unübersichtlich.
- Projektmappen Explorer: Hier sind alle Dateien des aktuell geöffneten Projekts aufgelistet. Über dieses Fenster lassen sich neue Dateien (gleich welcher Art) hinzufügen und die Projekteigenschaften bearbeiten.
- Eigenschaftsfenster: Hier können die Eigenschaften des im Designer gewählten Objektes verändert werden.
- Designer ist die Zeichenfläche des GUI-Builders.

Bedienung:

Die Bedienung beim Einfügen von Komponenten wird durch die Möglichkeit der absoluten Positionierung am Raster vereinfacht. Der GUI-Builder bietet mehrere Möglichkeiten, GUI-Komponenten auf den Designer zu platzieren.

- Durch Drag & Drop können Elemente aus der Toolbox auf den Designer gezogen werden. Hierbei kann die Größe des Elements bereits beim Erzeugen angepasst werden.
- Nach einem Doppelklick auf die Komponente in der Toolbox erscheinen die bereits angeklickten GUI-Komponenten auf dem Designer.
- Nach einem Klick auf die Komponente in der Toolbox und einen weiterer Klick auf den Designer wird die Komponente an der gewünschten Stelle eingefügt.

Codegenerierung:

Es wird Code in mehreren Sprachen, u. a. in Visual Basic, C#, J# und C++, erzeugt. Der erzeugte Code ist gut lesbar und übersichtlich. Wird eine GUI-Komponente gelöscht, so werden alle betreffenden Codestellen dieser Komponente, abgesehen von den „Event-Methoden“, gelöscht. Dies hat den Vorteil, dass in diesen „Event-Methoden“ einprogrammierter Code beim Löschen bzw. Verändern der GUI nicht verloren geht. Anderseits kann es dazu führen, dass im Code noch überflüssige Methoden erhalten bleiben, die nirgends mehr aufgerufen werden. Diese Methoden müssen vom Entwickler manuell gelöscht werden, was wiederum mehr Aufwand mit sich bringt. Der generierte Code enthält Kommentare, die zu besserer Nachvollziehbarkeit des Codes führen.

2.4.3 Folgerung

Die untersuchten GUI-Builder Jigloo und das Produkt von Visual Studio.Net weisen unterschiedliche Stärken und Schwächen auf.

Die Vorteile von Eclipse sind, dass es frei verfügbar und sehr gut für die Entwicklung von Java-Programmen geeignet ist. Visual Studio.Net hingegen ist nicht frei benutzbar.

Die Besonderheit von Visual Studio.Net liegt eindeutig in der Unterstützung mehrerer Programmiersprachen. So können Anwender in der Programmiersprache ihrer Wahl entwickeln. Beide Produkte bieten eine einheitliche Oberfläche. Die Oberfläche ist durch die klare Aufteilung in mehrere voneinander abgegrenzte Komponenten anschaulich. Sie ist nicht mit unnötigen Details überladen.

Die Palette ist sowohl bei Jigloo als auch beim GUI-Builder von Visual Studio gut erreichbar.
Die Bedienung des GUI-Builders in Visual Studio ist sehr praktisch. Hier werden im Gegensatz zu Jigloo mehrere Operationen zum Einfügen von GUI-Komponenten auf dem Designer unterstützt. Ein weiterer Vorteil in der Bedienung bei Visual Studio liegt eindeutig in der Möglichkeit, die Größe einer GUI-Komponente direkt beim Erzeugen beliebig zu definieren, so dass sie nicht nachträglich geändert werden muss.

2.4.4 Fazit

Die Ergebnisse der Analyse haben dazu geführt, dass wir uns für Visual Studio entschieden haben. Der Hauptgrund ist die Flexibilität, die durch die Möglichkeit der Programmierung und der Codegenerierung in mehreren Sprachen gegeben ist. Der Anwender kann die

Programmiersprache wählen, die für eine Problemlösung am besten geeignet ist; er ist in Grenzen nicht an eine bestimmte Programmiersprache gebunden.

Außerdem ist der GUI-Builder in Visual Studio ein fester Bestandteil der Entwicklungsumgebung und erfordert somit keine weiteren Installationen. Darüber hinaus bietet der GUI-Builder in Visual Studio dem Anwender mehr Möglichkeiten bei der Bedienung. Der Anwender ist somit nicht an eine bestimmte Vorgehensweise zum Erzeugen von GUI-Komponenten gebunden.

2.5 Zustandsdiagramm-Editoren

Im Rahmen dieser Analyse haben wir uns auf die Zustandsdiagramm-Editoren von Together [Together] und MS Visio [Visio2003] beschränkt. Die große Funktionsfülle von Together und Visio bietet eine Vielzahl von Gesichtspunkten, um neue Anforderungen zu gewinnen.

2.5.1 Together 6.2

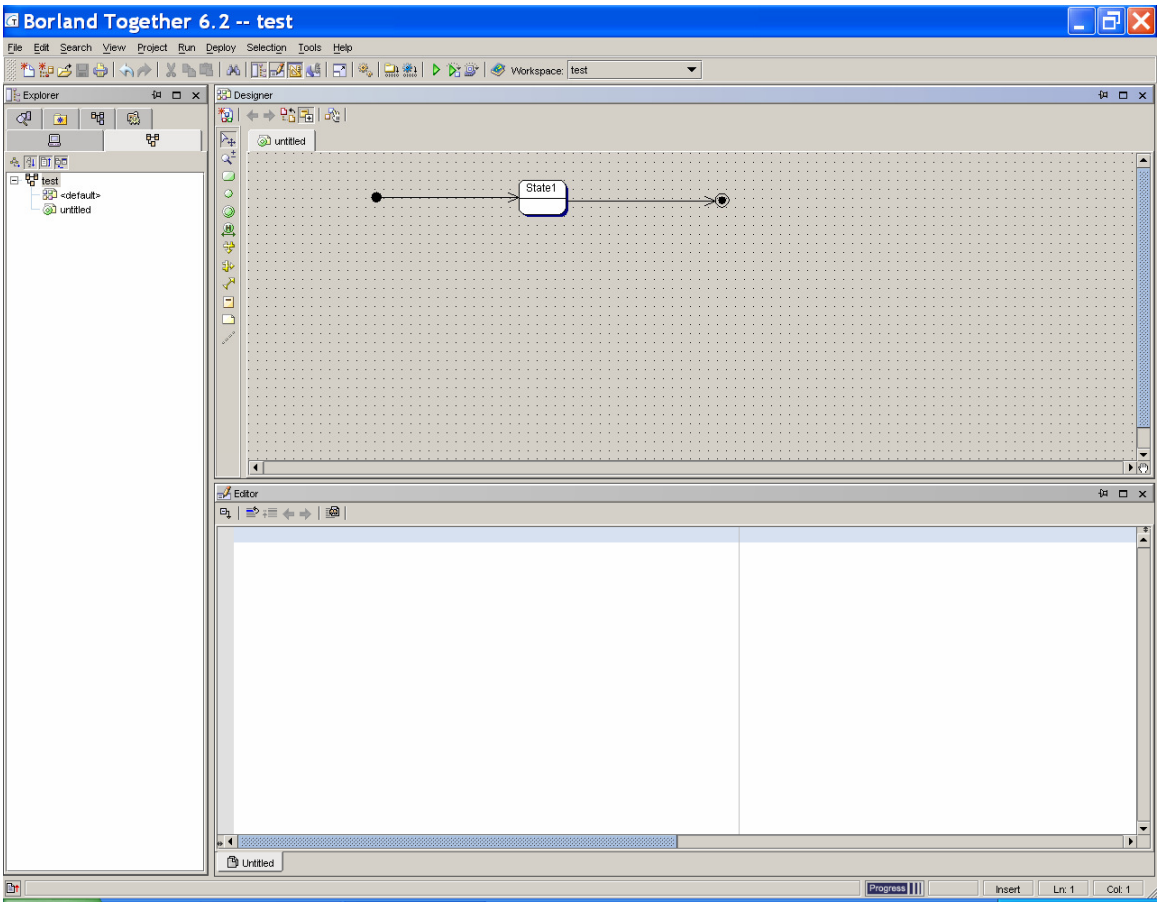


Abb. 15 Together

Together ist ein kostenpflichtiges UML-Modellierungstool von Borland. Es ist ein mächtiges Tool, das mehrere Diagrammtypen unterstützt und dem Entwickler Code-Generierung in mehreren Programmiersprachen bietet.

Im Rahmen unserer Bachelor-Arbeit betrachten wir jedoch nur die Modellierung in Form von Zustandsdiagrammen in Together. Der Zustandsdiagrammeditor ist sehr benutzerfreundlich. Rastermuster dienen der Orientierung beim Zustandsmodell-Entwurf. Es bietet u.a. Notizfelder zum Hinzufügen von Kommentaren, „Description“ als Properties für einen Zustand als weiterer Kommentarmöglichkeit an etc.

Praktisch ist auch das Anlegen von Zuständen, da die Größe und Position bereits während der Platzierung per Drag and Drop festgelegt werden kann. Um Zustandsdiagramme möglichst übersichtlich und aussagekräftig zu gestalten, können Zustände in verschiedenen Farben dargestellt werden. Zustände können ineinander verschachtelt werden. Nachteilig beim Zustandsdiagrammeditor von Together sind die hohen Ansprüche an die Hardware. Außerdem stellt er derart viele Funktionalitäten zur Verfügung, dass der Überblick leicht verloren gehen kann. Ein weiterer, wichtiger Nachteil besteht darin, dass Together sogar bei der Verwendung für nichtkommerzielle Zwecke kostenpflichtig ist. Der Zustandsdiagrammeditor von Together bietet keine Druckansicht.

2.5.2 MS Visio

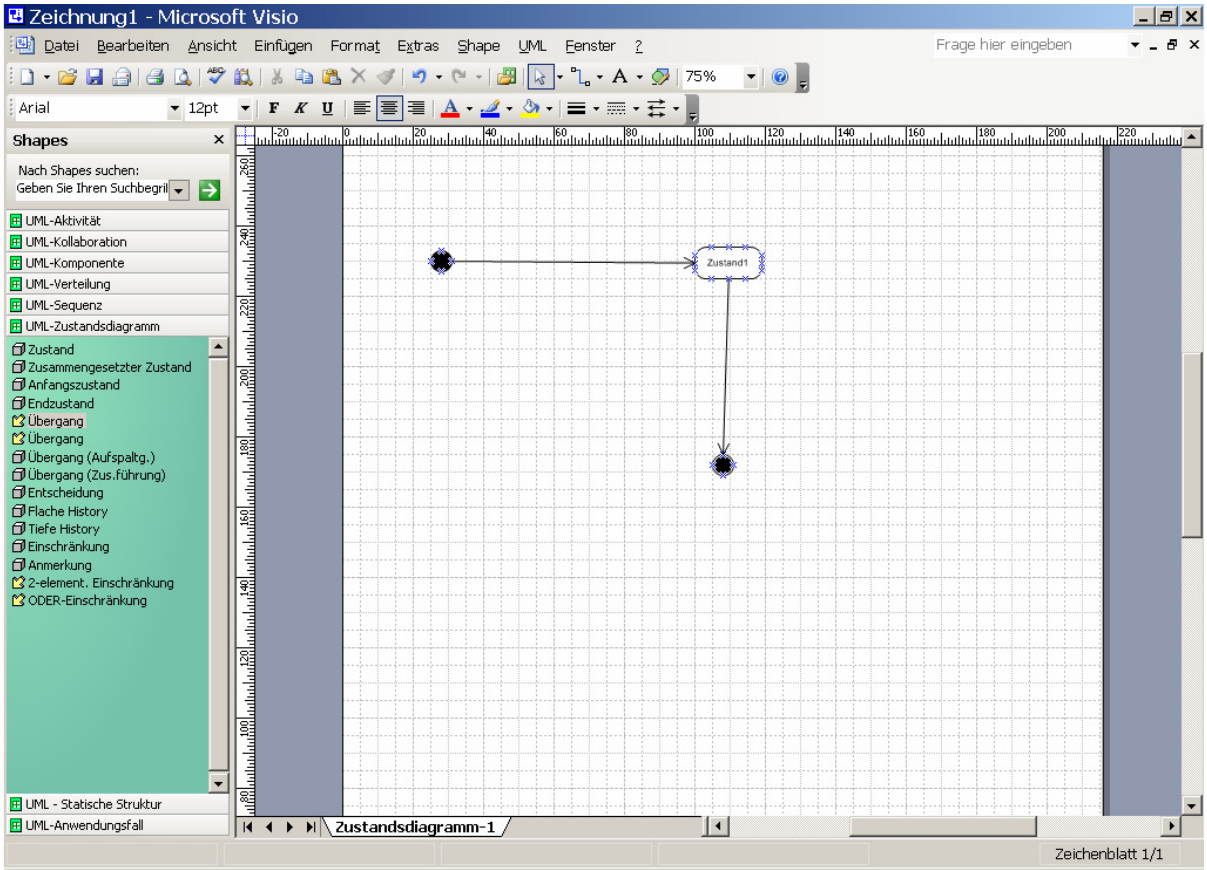


Abb. 16 Visio

Microsoft Visio 2003 ist ein Bestandteil des Microsoft Office Systems. Es bietet dem Anwender die Möglichkeit, Informationen in Form verschiedener Diagrammtypen abzubilden. Im Rahmen unserer Bachelor-Arbeit beschränken wir uns jedoch auf die Modellierung in Form von Zustandsdiagrammen in Microsoft Visio.

Der Zustandsdiagrammeditor bietet eine sehr benutzerfreundliche Zeichenumgebung. Gitter dienen der Positionierung von Zustandsdiagrammelementen. Zustandsdiagrammelemente können somit vertikal oder horizontal ausgerichtet werden, indem man sich am Gitter des Zeichenblattes orientiert.

Anders als in Together erhält der Benutzer die Möglichkeit, seine Seite individuell einzurichten. Die Größe des Zeichenblattes kann vom Anwender beliebig definiert werden. Die Seite kann im Querformat oder Hochformat dargestellt werden.

Der Benutzer kann außerdem noch die Ränder des Zeichenblattes ändern. Die Menüs und Symbolleisten in Visio sind mit den Menüs und Symbolleisten in anderen Microsoft Office-Programmen vergleichbar, so dass man standardisierte Verfahren zum Öffnen, Drucken und Speichern von Diagrammen benutzen kann.

Zustandsdiagrammelemente können auf vielfältige Weise hervorgehoben werden: durch die Farbe, das Muster, die Farbe der Linien, die das Muster bilden, die Stärke der Linien etc.

Außerdem stellt der Zustandsdiagrammeditor von Visio wie alle anderen Microsoft Office-Programme verschiedene Möglichkeiten zum Formatieren von Text zur Verfügung.
Ein weiterer Vorteil besteht in der Möglichkeit des Hinzufügens von Hyperlinks zu Zustandsdiagrammelementen, denn mit diesen Verknüpfungen kann man zu einer anderen Seite in demselben Diagramm, zu einem anderen Visio-Diagramm, zu einer beliebigen Datei oder zu einer Webseite wechseln.

Um eine einwandfreie Druckausgabe zu gewährleisten, bietet Visio eine Druckansicht.
Von großer Bedeutung ist auch die Möglichkeit, ein Diagramm in Microsoft Office Word-Dokumente zu integrieren und von dort aus direkt zu ändern.

Ein wesentlicher Nachteil von Visio liegt darin, dass Visio auch für nichtkommerzielle Zwecke nicht frei erhältlich ist.

2.5.3 Folgerung

Im Rahmen der Analyse der Zustandsdiagramm-Editoren haben wir eine größere Menge Anforderungen für die Eigenentwicklung eines Werkzeuges erkannt. Im Folgenden werden die in Together und Visio bereits identifizierten Anforderungen, die für die Realisierung des Tools dieser Bachelor-Arbeit sinnvoll sind, aufgelistet.

Jede Anforderung wurde nach den Kriterien must have, should have oder nice to have klassifiziert.

Abkürzung	**Beschreibung der Anforderung**	**Erkannt in**	**Klassifikation**
Rastermuster	Rastermuster zum Positionieren von Zustandsdiagrammelementen	Together	Should have
Notizfelder	Kommentare oder Beschreibungen können in Notizfelder eingetragen werden	Together	Nice to have

Description-Property	Jeder Zustand hat einen Description-Property, in dem zustandsspezifische Informationen eingetragen werden können	Together	Should have
Platzieren von Zuständen per Drag&Drop	Zustände können per Drag&Drop-Verfahren auf der Zeichenfläche platziert werden	Together	Should have
Platzieren von Zustandsübergängen per Drag&Drop	Zustandsübergänge können per Drag&Drop Verfahren auf die Zeichenfläche platziert werden	Together	Should have
Größe von Zuständen beim Erzeugen ändern.	Die Größe kann direkt beim Platzieren per Drag&Drop geändert werden.	Together	Should have
Farbliche Hervorhebung	Zustände können in verschiedenen Farben dargestellt werden.	Together / Visio	Nice to have
Größe des Zeichenblattes ändern	Das Zeichenblatt kann verkleinert oder vergrößert werden.	Visio	Nice to have
Format des Zeichenblattes	Es kann zwischen einem Querformat oder einem Hochformat ausgewählt werden	Visio	Nice to have
Formatieren von Text.	Text und andere Beschriftungen können beliebig formatiert werden	Visio	Nice to have
Formatierung – Muster	Für jedes Element kann eine Füllmuster definiert werden	Visio	Nice to have
Formatierung – Linie	Linienbreite, Linienfarbe kann für jedes Element beliebig definiert werden	Visio	Nice to have
Druckansicht	Eine Druckansicht kann vor dem Drucken angezeigt werden.	Visio	Nice to have

Tabelle 2 Neu erkannte Anforderungen aus Visio und Together

2.6 Fazit

Die Erzeugung von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen ist in der Literatur bisher noch nicht beschrieben worden. Die Recherche nach einem Werkzeug, das die in 1.2.1 beschriebene Methode umsetzt ergab dementsprechend kein Ergebnis.

Es sind zwar GUI-Builder zur Gestaltung graphischer Oberflächen oder Zustandsdiagramm-Editoren zum Erstellen von Zustandsdiagrammen, verfügbar. Aber ein Werkzeug, das die Funktionalität eines GUI-Builders und eines Zustandsdiagramm Editors kombiniert, wurde im Rahmen unsere Recherche nicht gefunden. Daraus leitet sich die Notwendigkeit ab, SUIT zu realisieren.

Die Implementierung eines eigenen GUI-Builders wird nicht geplant (siehe 1.2.3). Es wurden deshalb die GUI-Builder Jigloo und Visual Studio.Net GUI-Builder näher betrachtet. Die

Analyse hat dazu geführt, dass wir uns für den integrierten GUI-Builder von Visual Studio.Net als geeigneter GUI-Builder zur Wiederverwendung für das Tool dieser Bachelor-Arbeit entschieden haben. Weiterhin wurden die Zustandsdiagramm Editoren von Together und Visio betrachtet, um neue sinnvolle Anforderungen für die Realisierung eines eigenen Zustandsdiagrammeditor in SUIT finden.

3. Theorie

3.1 Einleitung

Den Schwerpunkt dieses Kapitels stellt die Erzeugung von Benutzungsschnittstellen mit Hilfe eines endlichen Automaten und Fensterentwürfen dar.

In Abschnitt 3.2 wird dieser Ansatz detaillierter erläutert und das im Rahmen dieser Bachelor-Arbeit zu implementierende Tool vorgestellt.

Eine Möglichkeit, welche in [Denert 1992] aufgezeigt wird, ist die Modellierung von Benutzungsschnittstellen in Form von Interaktionsdiagrammen. Diese Vorgehensweise wird im Rahmen dieses Kapitels in Abschnitt in 3.3 beschrieben.

In Abschnitt 3.4 wird die von uns im Rahmen dieser Bachelor-Arbeit verwendete Vorgehensweise, in Abgrenzung zur Vorgehensweise von Denert, dargestellt und diskutiert.

In Abschnitt 3.5 wird ein Fazit aus den Überlegungen gezogen.

3.2 Benutzungsschnittstellen auf Basis von endlichen Automaten und Fensterentwürfen

Die zustandsbasierte Modellierung stellt eine Möglichkeit dar, das Verhalten von Benutzungsschnittstellen zu spezifizieren. Die Benutzungsschnittstelle eines Systems kann als eine endliche Menge von Zuständen und Zustandsübergängen modelliert werden. In Abhängigkeit von einer bestimmten Eingabe wechselt der Zustandsautomat von einem Zustand in einen anderen. Welche Eingaben möglich sind, ist vom aktuellen Zustand des Systems abhängig. Eingaben verursachen Zustandsübergänge und dienen dem Benutzer zum Navigieren in den endlichen Automaten.

Eine Benutzungsschnittstelle besteht in der Regel aus einer Menge von Masken, zwischen denen durch Eingabeaktionen (Menüauswahl, »OK«-Button usw.) hin und her gewechselt werden kann. Werden Masken bzw. Dialogboxen als Zustände aufgefasst und Eingabeaktionen als Kantenübergänge, so können die Navigationsmöglichkeiten durch die Bedienoberfläche als endlicher Automat modelliert werden. In jedem Zustand wartet das System auf eine Benutzereingabe. Ein Mausklick auf einen Button z.B. ist ein Ereignis, das eine Aktion auslöst, die einen Zustandswechsel in den Folgezustand bewirken kann.

Bei der Modellierung einer Benutzungsschnittstelle eines Systems in Form eines endlichen Automaten kann daher jedem Zustand eine Maske zugeordnet werden.

Für eine Anwendung dieses Ansatzes ist eine über zeichnerische Modelleditoren hinausgehende Unterstützung durch ein Werkzeug sinnvoll. Ein Schritt in diese Richtung ist die Implementierung eines graphischen Editors, der die zustandsbasierte Modellierung von

Benutzungsschnittstellen unterstützt. Der Editor sollte darüber hinaus über einen integrierten GUI- Builder zur Gestaltung der graphischen Oberfläche verfügen. Auf dieser Überlegung basiert die Idee zu SUIT. Dieses Werkzeug bereichert die Visual Studio-Umgebung um einen Zustandsdiagrammeditor mit dem integrierten GUI-Builder von Visual Studio, eine Simulationskomponente und einen Codegenerator (Siehe 1.2).

SUIT bietet damit die Möglichkeit, das dynamische Verhalten von Benutzungsschnittstellen auf Basis von Zustandsdiagrammen und Fensterentwürfen zu modellieren, das entstandene Modell zu simulieren und aus dem Modell das Codegerüst für eine Fassade in einer .Net-Sprache zu generieren.

Die Simulationsmöglichkeit ermöglicht eine explorative Vorgehensweise. Ziel ist es, das Systemverhalten unter dynamischen Aspekten aufzuzeigen. Es wird ein Prototyp als Kommunikationsmedium entwickelt, der dazu dienen soll, die Anforderungen an das zu implementierende System zu verstehen und zu verfeinern. Durch die Visualisierung anhand der Simulation erhält der Anwender ein Bild des zukünftigen Systems und kann somit frühzeitig korrigierend eingreifen.

SUIT ermöglicht auch die Weiterentwicklung von Prototypen bis zur einsatzfähigen Anwendung, da das Tool qualitativ hochwertigen Code erzeugt, der nicht verworfen wird, sondern bedarfsweise wieder verwendet werden kann. SUIT erzeugt eine abstrakte Schnittstelle, die Fassade. Durch die Trennung der Interaktionskomponente und der Funktionskomponente durch eine Fassade wird ein sauberes nachvollziehbares Design unterstützt (siehe Abschnitt 1.2).

3.3 Vorgehensweise mit Interaktionsdiagrammen

Im Folgenden wird die Vorgehensweise von Denert beschrieben [Denert 1992]. Als Beschreibungsmethode zur Modellierung des dynamischen Verhaltens einer Benutzungsschnittstelle dienen hier Interaktionsdiagramme (IAD), eine Variante endlicher Automaten. Es wird jedoch nicht auf alle Details eingegangen, die in [Denert 1992] bei der Spezifikation einer Benutzungsschnittstelle berücksichtigt werden.

3.3.1 Interaktionsdiagramme

Interaktionsdiagramme sind eine Variante endlicher Automaten. Die wesentlichen Elemente eines Interaktionsdiagramms sind die Zustände und die Zustandsübergänge. In Interaktionsdiagrammen werden die Zustände als Kreise visualisiert. Der Zustand markiert einen Punkt im Dialogablauf, in dem das System auf eine Eingabe des Benutzers wartet. Der Startzustand wird im Interaktionsdiagramm durch ein Dreieck visualisiert. Die Zustandsübergänge werden durch Pfeile zwischen den Zuständen dargestellt. Die Pfeile sind mit virtuellen Tasten (siehe Abschnitt 3.3.2) beschriftet.

Aktionen werden dadurch ausgelöst, dass ein Übergang stattgefunden hat. Zwischen zwei Zuständen können eine bis mehrere Aktionen ausgeführt werden. Im Unterschied zu den endlichen Automaten werden Aktionen in einem Interaktionsdiagramm als eigenständige Elemente in Form eines Rechteckes dargestellt. Eine Aktion kann mehrere Ausgänge besitzen. Jeder Ausgang ist mit einem Aktionsergebnis versehen.

Im Folgenden wird in Abb. 17 das Interaktionsdiagramm eines Geldausgabeautomaten aus [Denert 1992] beispielhaft dargestellt:

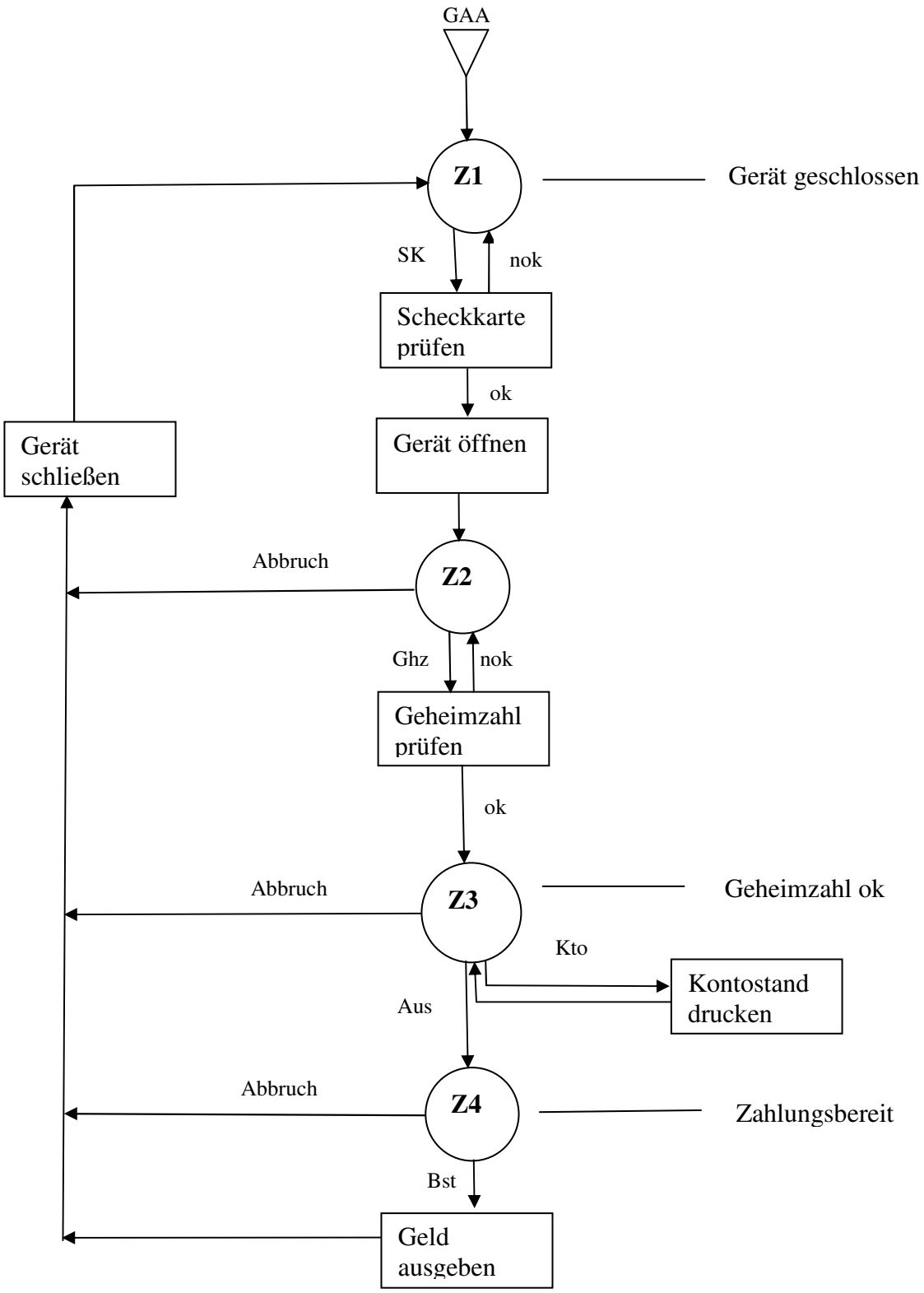


Abb. 17 Interaktionsdiagramm eines Geldausgabeautomaten aus [Denert 1992]

Neben der graphischen Notation in Form von Interaktionsdiagrammen benutzt Denert aus Gründen der Übersichtlichkeit einen Interaktionsdiagramm-Code, um das Modell exakter zu

spezifizieren und es nicht mit Details und Sonderfällen zu überladen. Im Interaktionsdiagramm-Code werden alle Zustände des Interaktionsdiagramms, angefangen vom Initialzustand, aufgelistet. Für jeden Zustand werden alle ausgehenden Zustandsübergänge erfasst und die durch sie angestoßenen Aktionen sowie der Folgezustand beschrieben.

Ein Interaktionsdiagramm eignet sich zwar dafür, die wesentlichen Abläufe verständlich, nachvollziehbar und bildhaft zu visualisieren. Werden jedoch zu viele Details und Sonderfälle in einem Interaktionsdiagramm dargestellt, kann es, wie bereits erwähnt, schnell unverständlich und unübersichtlich werden.

Für das Interaktionsdiagramm des Geldautomaten in Abb. 17 sieht der von Denert verwendete Code [Denert 1992] wie folgt aus:

Zustand Z1 ---Gerät geschlossen
SK: Scheckkarte prüfen
Falls ok: Gerät öffnen → Z2
Falls nok: --> Z1

Zustand Z2 ---Scheckkarte geprüft
Ghz: Geheimzahl prüfen
Falls ok --> Z3
Falls nok --> Z2
Abb: Gerät schließen --> Z1

Zustand Z3 – Geheimzahl ok
Kto: Kontostand drucken → Z3
Aus: → Z4
Abb: Gerät schließen → Z1

Zustand4 –zahlungsbereit
Bst: Geld ausgeben
Gerät schließen → Z1
Abb:Gerät schließen → Z1

In Z1 z.B. ist nur die virtuelle Taste (siehe Abschnitt 3.3.2) SK zulässig. Das Betätigen dieser Taste stößt die Aktion „Scheckkarte prüfen“ an. Wurde die Aktion erfolgreich ausgeführt (ok), wird die nächste Aktion „Gerät öffnen“ angestoßen; das System gelangt in den Folgezustand Z2. Ansonsten kehrt das System zu Z1 zurück.

3.3.2 Virtuelle Tasten

Das Konzept der virtuellen Tasten beinhaltet jegliche Art „dialogsteuernder Information“[Denert 1992]. Die physische Realisierung einer virtuellen Taste kann unterschiedlich sein. Virtuelle Tasten lösen Zustandübergänge aus und ermöglichen somit das Navigieren durch das Interaktionsdiagramm.

Diese kann erfolgen durch:

- Das Betätigen einer Funktionstaste
- Durch Menüauswahl
- Durch Eingabe eines Kommandos in einem bestimmten Feld
- Oder „implizit durch Dateninhalt; die Steuerinformation liegt dabei meist in der Unterscheidung, ob der Schlüssel eines Datenobjekts und / oder der zugehörige Fachinhalt eingegeben wurde oder nicht“ [Denert 1992, Seite 132]

Denert unterscheidet zwischen globalen virtuellen Tasten und speziellen virtuellen Tasten. Die speziellen virtuellen Tasten sind nicht allgemein gültig. Sie werden in einer Entscheidungstabelle spezifiziert.

Globale virtuelle Tasten sind Virtuelle Tasten, die unabhängig vom aktuellen Zustand immer dasselbe bewirken - wie z.B. das Vor- und Rückwärtsblättern in Informationen oder den Aufruf der Hilfe. Sie können einmalig systemweit definiert werden. In der Tabelle der globalen virtuellen Tasten werden die Standard-Interaktionen und ihre Realisierung beschrieben; es wird festgehalten, welche Aktionen sie anstoßen und in welchem Folgezustand sie das System überführen. Auf diese Weise müssen wiederkehrende virtuelle Tasten nicht in jedem Zustand neu definiert werden; die Interaktionsdiagramme werden somit nicht mit redundanten Informationen überladen.

3.3.4 IAD- Zustände und Masken

Bei der Modellierung einer Benutzungsschnittstelle in Form von Interaktionsdiagrammen kann jedem Zustand eine Maske zugeordnet werden. Die Zuordnung wird in einer Maskentabelle erfasst.

3.3.5 Dialogwechsel

Wird ein Dialog verlassen, um zu einem anderen zu wechseln, bestehen bzgl. der Behandlung des verlassenen Dialogs zwei grundsätzliche Alternativen:

1. Die in ihm erfassten Daten werden verworfen oder bestätigt; dadurch ist er abgeschlossen.

2. Wird der Dialog weder verworfen noch bestätigt, bleibt er offen. Die Bearbeitung der Daten ist nur unterbrochen oder suspendiert. Daher ist eine Rückkehr zu ihm möglich.

Wird der Dialog unterbrochen (durch einen Dialogwechsel), so wird der jeweils aktuelle Zustand zwischengespeichert. Kehrt der Anwender zu ihm zurück, befindet sich der Dialog weiterhin in genau diesem gespeicherten Zustand.

Eine spezielle Variante besteht darin, einen Dialog zu unterbrechen, um eine neue Inkarnation desselben Dialogs mit anderen Daten zu ermöglichen.

Die Anzahl der Inkarnationen eines Dialogs kann festgelegt werden:

1. Ist sie = 0, wird der Dialog bei einem Wechsel zu einem anderen Dialog abgeschlossen. Der aktuelle Zustand wird nicht gespeichert; bei einem erneuten Aufruf wird daher der Initialzustand wiederhergestellt.

2. Ist sie maximal = 1, wird immer nur die zeitlich letzte Inkarnation gespeichert, so dass eine Rückkehr zu diesem Dialog möglich ist.

3. Ist sie > 1, ist die Festlegung einer Regel erforderlich, nach der zu einem bestimmten Dialog zurückgekehrt wird. Mögliche Alternativen sind:

- die zeitlich letzte Inkarnation (LIFO-Kellerprinzip) wird gewählt
- die erste Inkarnation (FIFO-Warteschlangenprinzip) wird gewählt
- die Wahl ist durch den Anwender vorzunehmen; daher muss ihm eine Selektionsmöglichkeit angeboten werden.

Beim Übergang von einem Dialog zu einem anderen sind bzgl. ihrer Behandlung vier Möglichkeiten gegeben:

- der alte Dialog wird verlassen und ein neuer begonnen
- der alte Dialog wird verlassen und ein offener wieder aufgenommen
- der alte Dialog wird unterbrochen und ein neuer begonnen
- der alte Dialog wird unterbrochen und ein offener wieder aufgenommen

Diese Möglichkeiten müssen nicht vollständig realisiert bzw. erlaubt sein. Zwar ist meist ein flexibler Wechsel zwischen Dialogen wünschenswert; aber es kann erforderlich sein, den Zugang zu einem Dialog nicht von jedem anderen aus zu ermöglichen, sondern jeweils zu beschränken. Dies geschieht dadurch, dass er an Bedingungen geknüpft wird (z.B. indem ein Passwort oder für den zweiten Dialog benötigte Daten übergeben werden).

3.3.6 IAD-Aktionen

Bei IADs werden außerdem die Aktionen einbezogen, die durch das Betätigen der virtuellen Taste angestoßen werden. Wie sind diese Aktionen zu spezifizieren? Da in den Aktionen dialogspezifische Daten bearbeitet werden, kann dies nur für den einzelnen Dialog erfolgen.

Um eine Aktion zu spezifizieren, ist der spezifische Dialog auf die von ihm auszulösende Aktion hin zu analysieren. Ausgangspunkt sind die verschiedenen Felder und Feldinhalte des Dialogs, für den eine Aktion ausgelöst wird. In bestimmten Feldern werden Daten neu eingegeben, geändert oder gelöscht: Dies bildet die Grundlage für die Bearbeitung der entsprechenden Geschäftsvorfälle, durch welche wiederum mittelbar oder unmittelbar Sachbearbeiter mit Teilaufgaben beauftragt werden.

Ef1	n	n	ä			...	Ef Eingabefeld
Ef2	n			Ä			
...				Ä			Gv Geschätfsvorfall
...							n neu eingegeben
...							ä geändert
Efm		n			L	...	l gelöscht
Gv1	x					...	
Gv2		x					
...					x		
...				x			
...			x				
Gvn			x			...	

Abb. 18 Spezifikation einer Aktion mittels Entscheidungstabelle [Denert 1992]

Um die Abhängigkeiten von Feldinhalten und Geschäftsvorfällen bzw. Sachbearbeiteraufträgen übersichtlich darzustellen, kann die Entscheidungstabelle verwendet werden (siehe Abb. 18). Für jede Aktion eines Dialogs muss eine eigene Spezifikation erfolgen.

Die Entscheidungstabelle für IAD-Aktionen Abb. 18 enthält zwei Teile: den Bedingungs- und den Aktionsteil.

Im Bedingungsteil (obere Hälfte) werden die Felder untereinander aufgelistet, deren Inhalte in der Aktion zu bearbeiten sind. Es wird nun spaltenweise analysiert, welche verschiedenen Eingaben (neu eingegeben, geändert, gelöscht) für diese Felder im Hinblick auf eine Aktion zusammenhängen können.

Im Aktionsteil (untere Hälfte) werden die potentiell zu bearbeitenden Geschäftsvorfälle (evtl. auch die unmittelbar anzustoßenden Sachbearbeiteraufträge) untereinander aufgelistet. Für jeden wird festgelegt, von welchen Bedingungen (Eingaben im oberen Teil) seine Aktivierung abhängt.

Eine IAD-Aktion kann unterschiedliche Ergebnisse herbeiführen. Im IAD selbst werden nur Bezeichnungen für diese angegeben (oder nichts, falls nur das Ergebnis „o.k." möglich ist).
Auch die Ermittlung dieser Ergebnisse kann nur für jeden einzelnen Dialog durchgeführt werden. Diese Spezifizierung kann ebenfalls mithilfe einer Entscheidungstabelle vorgenommen werden.

Der Vorteil einer Entscheidungstabelle zur Analyse von Benutzereingaben besteht vor allem darin, dass sie zur vollständigen Analyse zwingt, da alle theoretisch möglichen Eingaben und ihre Kombinationen dargestellt werden. Bestimmte Kombinationen werden andernfalls leicht übersehen, da sie als unmöglich oder sinnlos eingeschätzt werden. Zwingend ist der Einsatz einer Entscheidungstabelle jedoch nicht; gelegentlich ist eine verbale Darstellung oder eine Darstellung mittels Pseudocode einfacher und leichter nachzuvollziehen.

3.4 Die in dieser Arbeit gewählte Vorgehensweise

3.4.1 Vorgehensweise mit Zustandsdiagrammen

Anders als Denert, der Interaktionsdiagramme für die Beschreibung von Benutzungsschnittstellen wählt, wird in dieser Arbeit die Modellierung in Form von Zustandsdiagrammen vorgenommen.

Zustandsdiagramme basieren wie die von Denert beschriebenen Interaktionsdiagramme auf endlichen Automaten. Die Basiselemente eines Zustandsdiagramms sind Zustände und ihre Übergänge (Transitionen). Zustandsdiagramme beschreiben das Verhalten eines Systems als eine Menge von Zuständen und Zustandsübergängen. Ein Zustand beschreibt das System zu einem bestimmten Zeitpunkt. Ein Zustandsübergang ist ein Zeitpunkt, an dem das System einen gegebenen Zustand verlässt und in einen neuen Zustand übergeht. Er ist durch ein Ereignis und eine Aktion spezifiziert. Aktionen werden im Zustandsdiagramm dadurch ausgelöst, dass ein Übergang stattgefunden hat.

Im Rahmen dieser Bachelor-Arbeit wurden flache Zustandsdiagramme für die Modellierung gewählt. Die praktische Begründung dafür liegt in dem Vorhaben, diese im Rahmen der nächsten Version auf Statecharts zu erweitern, sofern sich die Erzeugung von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen mit SUIT als praxistauglich erweist.

Statecharts wurden von Harel entwickelt; sie erweitern Zustandsdiagramme um Hierarchieebenen und Parallelität. Sie wurden ursprünglich informell vorgestellt. Eine erste formale Beschreibung erfolgte dann in [Harel 1987].

Zustandsdiagramme können sich bei der Modellierung umfangreicher Benutzungsschnittstellen als problematisch erweisen, da sie keine Hierarchie und keine Modularität ermöglichen. Statecharts erweitern Zustandsdiagramme unter anderem um Hierarchie und erleichtern somit das Modellieren dynamischen Verhaltens von Systemen. Die Einführung von Hierarchie in Statecharts verhindert, dass ein Modell bei komplexen Systemen unübersichtlich wird.

Die Notation von Harel ist bereits weit verbreitet und wird in der Softwareentwicklung eingesetzt, wie z.B. in UML. Einige Statechart Editoren wurden bereits in der von uns durchgeführten Marktanalyse untersucht. Die Realisierung einer Modellierung in Form von Statecharts würde aber den Rahmen der Bachelor-Arbeit sprengen und wird deshalb für die nächste Version geplant.

Denert berücksichtigt bei der Spezifikation von Benutzungsschnittstellen eine große Anzahl Details und versucht diese zu einem sinnvollen Ganzen zu verbinden [Denert 1992]. Wir konzentrieren uns schwerpunktmäßig auf die Erzeugung von Benutzungsschnittstellen-Prototypen mit Hilfe von Zustandsdiagrammen und Fensterentwürfen und die automatische Generierung einer Fassadenschnittstelle (siehe Abschnitt 1.2).

Die Erzeugung der Prototypen soll mit möglichst geringem Aufwand erfolgen. Sie sollten leicht änderbar und bei Bedarf zu einem ausgereiften System erweiterbar sein. Das Hauptziel

dabei ist das dynamische Verhalten von Benutzungsschnittstellen in einer relativ frühen Phase eines Projektes aufzuzeigen und somit eine frühe Kundeneinbindung zu erreichen. Dadurch können das Anwendungsprobleme und mögliche Lösungen besser untersucht und verstanden werden, so dass Fehler früher als bisher entdeckt und beseitigt werden können und somit weniger Kosten verursachen.

Die Spezifikation einer Benutzungsschnittstelle besteht aus einer Vielzahl einzelner Elemente wie z.B. der Handhabung des Dialogwechsels (siehe Abschnitt 2.2.5) und der Analyse der Benutzereingaben etc. Alle Elemente zu berücksichtigen ist, wie bereits erwähnt, nicht unser primäres Ziel.

Wir verzichten z.B. auf die Auswertung von Benutzereingaben, wie sie Denert in Abschnitt 2.2.6 vornimmt. Die Simulation eines Systems an seiner Benutzungsschnittstelle soll eine vertraute Anschauung der Bildschirmmasken und –abläufe bieten. Um das dynamische Verhalten einer Benutzungsschnittstelle in den frühen Phasen der Softwareentwicklung zu beschreiben, ist die Auswertung von Benutzereingaben nicht erforderlich, da zu diesem Zeitpunkt die dahinter stehende Funktionalität noch nicht detailliert bekannt ist.

Von uns werden berücksichtigt:

- Zustandsdiagramme: Diese beschreiben das dynamische Verhalten einer Benutzungsschnittstelle. Zustandsdiagramme bestehen aus Zuständen und Zustandsübergängen. Bei den Zuständen unterscheidet man zwischen normalen Zuständen und dem Anfangzustand.

- Aktivität: ist eine Operation, die beim Eintritt in einem Zustand ausgeführt werden. Es handelt sich hierbei um einen Verweis auf dem den Zustand zugeordneten, ausführbaren Maskencode.

- Ereignisse: dies stellt bei unserem Verfahren die Entsprechung zu den virtuellen Tasten Denerts dar. Durch ein Ereignis wird eine Information an das System übertragen. Ein Beispiel für ein Ereignis könnte z.B. das Betätigen einer Taste sein. Ereignisse lösen, wie die virtuellen Tasten bei Denert, Zustandsübergänge aus. Das System kann dabei in jedem Zustand auf verschiedene Ereignisse mit dem Wechsel in andere Zustände reagieren. Mit Hilfe von Ereignissen kann der Benutzer durch das Modell navigieren.

- Aktionen: dies sind Operationen, die während einer Transition von einem Zustand zum anderen ausgeführt werden. Eine Aktion wird angestoßen, wenn ein Ereignis ausgelöst wird. Die Spezifikation von Aktionen, wie sie Denert vornimmt (Abschnitt 2.2.6), wird von uns nicht durchgeführt, da diese über die reine Simulation einer Benutzungsschnittstelle hinausgehen würde.

Masken werden in Form von Aktivitäten den Zuständen eines Zustandsdiagramms zugeordnet. In jedem Zustand wartet das System auf eine Benutzereingabe. Erfolgt eine Benutzereingabe, so navigiert das System von einem Zustand zum Folgezustand.

Gelänge man durch eine Aktion in einen Endzustand, so wären keine weiteren Eingabe- und somit keine Navigationsmöglichkeiten mehr möglich, da ein Endzustand das Ende eines Systems symbolisiert. Dem Endzustand wird keine Maske zugeordnet, weil keine Eingabe

mehr erfolgen darf. Wenn man sich also im Endzustand befindet, ist keine Maske mehr aktiv. Alle Masken sind geschlossen und es ist keine mehr offen.

Wir verzichten somit auf die Definition eines Endzustands, da das Ende eines Systems auch durch das Schließen aller offenen Masken erreicht werden kann. Denn wenn keine Maske mehr offen ist, bedeutet das, dass man sich in keinem der definierten Zustände des erstellten Zustandsdiagramms mehr befindet. Das Ende eines Systems kann also auch dadurch erreicht werden, wenn alle Masken geschlossen werden und keine mehr offen ist.

Die Einsparung des Endzustands führt zur Einsparung vieler Zustandsübergänge. Die Transitionen, die zum Endzustand hinführen, fallen dadurch weg. Sie werden im Modell nicht mehr visualisiert. Die Einsparung dieser Zustandsübergänge erhöht die Übersichtlichkeit innerhalb des modellierten Zustandsdiagramms. Denn viele GUI-Systeme können von jedem Dialog aus beendet werden. Das würde in einem Zustandsdiagramm bedeuten, dass von jedem Zustand eine Transition zum Endzustand führt. Dadurch werden Zustandsdiagramme unübersichtlich und unüberschaubar. Das Weglassen eines Endzustands führt also zur Einsparung vieler Zustandsübergänge und verhilft zu erhöhter Übersichtlichkeit innerhalb des Zustandsdiagramms.

3.4.2 SUIT-Notation

Zustand

Ein Zustand (Abb. 19) wird durch ein Rechteck visualisiert. Er enthält seine Bezeichnung und eine Aktivität. Bei der Aktivität handelt sich hierbei um eine Referenz auf den ausführbaren Maskencode (siehe 1.2).

Abb. 19 Zustand

Ein *Startzustand* (Abb. 20) besteht aus drei Komponenten, ein gefüllter Kreis, einen Zustandsübergang und einen Zustand, wobei der Zustandsübergang vom gefüllten Kreis zum Zustand hineinführt.

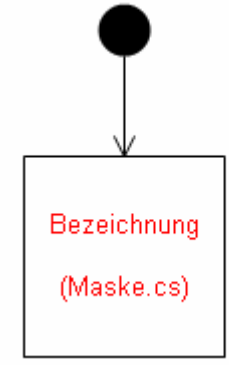


Abb. 20 Startzustand

Transition (Zustandsübergang)

Ein normaler Zustandsübergang (Abb. 21) wird als Pfeil zwischen zwei Zuständen visualisiert. Ein rekursiver Zustandsübergang (Abb. 22) führt zum Ausgangzustand zurück. Die Transitionsbeschriftung besteht aus einem Ereignis und einer Aktion (Ereignis/ Aktion)

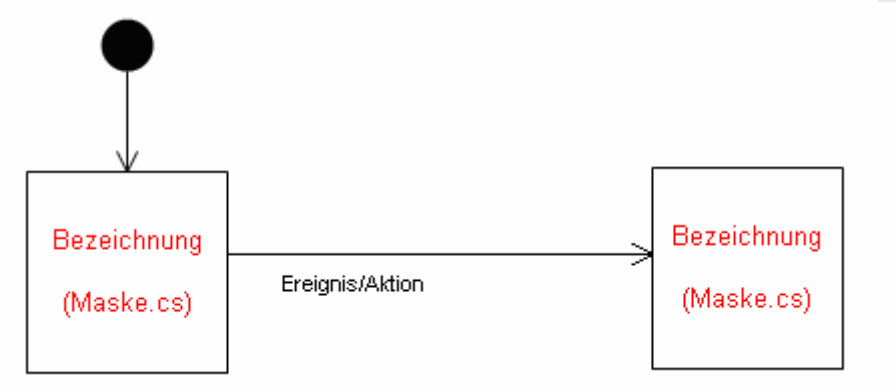


Abb. 21 Zustandsübergang

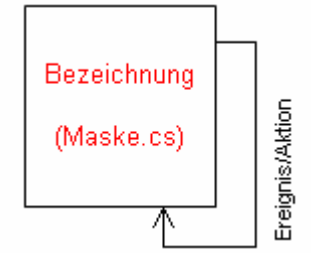


Abb. 22 Rekursiver Zustandsübergang

3.5 Fazit

Der zustandsbasierte Ansatz - die Modellierung von Benutzungsschnittstellen mithilfe endlicher Automaten und Fensterentwürfe - wird sowohl von Denert in seiner Vorgehensweise bei der Spezifikation einer Benutzungsschnittstellen als auch im Rahmen dieser Bachelor-Arbeit zur Modellierung einer Benutzungsschnittstelle verfolgt.

Denert berücksichtigt, wie bereits erwähnt, eine Vielzahl einzelner Elemente, die für die Spezifikation einer Benutzungsschnittstelle eigentlich unerlässlich sind. Einige davon wurden in Abschnitt 2.2 beschrieben. Er beschäftigt sich unter anderem mit der Handhabung eines Dialogwechsels und führt Entscheidungstabellen zur Analyse von Benutzereingaben ein, z.B. für die Spezifikation einer Aktion, wie sie in Abschnitt 2.2.6 beschrieben wurde.

Der Schwerpunkt bei der Entwicklung von SUIT liegt auf der schnellen Prototypenerstellung und der automatischen Generierung von Code in Form einer Fassade. Alle Details einer Benutzungsschnittstelle abzudecken steht daher bisher nicht im Zentrum der Aufmerksamkeit. Sollte sich die Modellierung von Benutzungsschnittstellen mithilfe von Zustandsdiagrammen und Fensterentwürfen jedoch als praxistauglich erweisen, würde die Erweiterung des Tools in dieser Richtung eine Bereicherung bedeuten.

4. Anforderungen

4.1 Einleitung

Das im Rahmen diese Bachelor-Arbeit zu implementierende Tool SUIT wird in die Entwicklungsumgebung Miscrosoft Visual Studio.NET integriert. SUIT verfügt nicht über einen eigenen GUI-Builder sondern nutzt die Funktionalitäten des GUI-Builders von Visual Studio. Die Implementierung eines eigenen GUI-Builder wird im Rahmen dieser Bachelor-Arbeit nicht geplant. In Abschnitt 4.2 werden alle funktionalen Anforderungen an SUIT beschrieben und klassifiziert. In 4.3 folgt ein Fazit.

4.2 funktionale Anforderungen

SUIT ist in drei Module aufgeteilt:

- ***Zustandsdiagramm-Editor***:
  Mit dem Zustandsdiagrammeditor werden das komplette dynamische Verhalten einer Benutzungsschnittstelle und die Gestaltung ihrer graphischen Oberfläche beschrieben.
  Der Zustandsdiagrammeditor ist der wesentliche Teil dieser Arbeit.
  Es handelt sich hierbei um einen graphischen Editor zum Editieren von Zustandsdiagrammen mit der Besonderheit, dass der Editor den integrierten GUI-Builder von Visual Studio nutzt.

- ***Simulator***:
  Wurde das dynamische Verhalten einer Benutzungsschnittstelle in Form eines Zustandsdiagramm modelliert und die Bildschirmmasken generiert, kommt die Simulationskomponente zum Einsatz.

- ***Codegenerator***
  Aus dem erstellten Zustandsdiagramm erzeugt der Codegenerator von SUIT eine Fassade der Funktionskomponente.

Im Folgenden werden tabellarisch (siehe Tabelle 2) alle funktionalen Anforderungen an SUIT aufgelistet. Jede Anforderung wird einer der folgenden drei Klassifikationen zugeordnet:

- ***Must have***, dies sind Anforderungen, die SUIT erfüllen muss. Sie werden im Rahmen dieser Bachelor-Arbeit realisiert.

- ***Should have***, dies sind mögliche Ergänzungen, die SUIT bereichern würden und im Rahmen dieser Bachelor-Arbeit, soweit es zeitlich möglich ist, realisiert werden.

- ***Nice to have***, dies sind mögliche Ergänzungen, die SUIT bereichern würden und im Rahmen dieser Bachelor-Arbeit, soweit es zeitlich möglich ist, realisiert werden. Wobei die ***Should have*** Anforderungen höher priorisiert sind als die ***Nice to have*** Anforderungen.

ID	**Abkürzung**	**Beschreibung der Anforderung**	**Klassifikation**
A1	Erzeugen einer Zustandsdiagrammklasse	Eine Zustandsdiagrammdatei wird erzeugt, die das Zustandsdiagramm des Anwenders beschreibt.	Must have
A2	Toolbox	Der Zustandsdiagramm - Editor verfügt über eine Toolbox, in der Zustände, Startzustände und Zustandsübergänge symbolisch angezeigt werden. Die Toolbox ist auf- und abklappbar.	Must have
A3	Erzeugen eines Zustandes per Drag & Drop	Ein Zustand kann per Drag & Drop von der Toolbox auf die Zeichenfläche platziert werden. Die Größe ist nicht vorgegeben und wird beim Erzeugen beliebig definiert.	Must have
A4	Erzeugen eines Anfangszustandes per Drag & Drop	Ein Anfangszustand kann per Drag & Drop von der Toolbox auf die Zeichenfläche platziert werden. Die Größe ist nicht vorgegeben und wird beim Erzeugen beliebig definiert.	Must have
A5	Erzeugen eines Zustandes per Doppelklick	Mit einem Doppelklick auf das Icon Zustand in der Toolbox wird ein Zustand fester Größe auf die Zeichenfläche platziert.	Should have
A6	Erzeugen eines Startzustand per Doppelklick	Mit einem Doppelklick auf das Icon Startzustand in der Toolbox wird ein Startzustand fester Größe auf die Zeichenfläche platziert.	Should have
A7	Erzeugen eines Zustandsübergang (Transition) per Drag&Drop	Ein Zustandsübergang wird per Drag & Drop von der Toolbox auf die Zeichenfläche platziert. Jeder Zustandsübergang muss einen hineinführenden und einen hinausführenden Zustand haben.	Must have
A8	Erzeugen eines rekursiven Zustandsübergangs	Rekursive Zustandsübergänge sollen erlaubt sein	Should have
A9	Löschen eines Zustandes	Ist ein Zustand selektiert, wird er beim Drücken der „Entf“ – Taste gelöscht. Die eingehenden und ausgehenden Zustandsübergänge werden gelöscht.	Must-have
A10	Löschen eines Startzustandes	Ist ein Startzustand selektiert, wird er beim Drücken der „Entf“ Taste gelöscht. Alle hinausgehenden Zustandsübergänge werden gelöscht.	Must Have

A11	Löschen eines Zustandsübergangs	Ist ein Zustandsübergang selektiert, wird er beim Drücken der „Entf“ Taste gelöscht.	Must have
A12	Zustandsübergänge Knickpunkte	In einem Zustandsübergang können Knickpunkte definiert werden, um eine gewinkelte Darstellung eines Zustandsübergangs zu ermöglichen.	Should have
A13	Notizfeld	Ein Notizfeld ist ein Zustandsdiagrammelement, das auch aus der Toolbox selektiert und frei auf die Zeichenfläche platziert werden kann	Nice to have
A14	Selektieren	Ein Anfangszustand, Zustand oder Zustandsübergang kann mit der Maus selektiert werden	Must have
A15	Selektieren des gesamten Zustandsdiagramms	Ein Zustandsdiagramm kann mit der Maus selektiert werden.	Should have
A16	Undo/ redo	Es gibt eine Undo-Funktionalität, die den letzten Bearbeitungsschritt rückgängig macht und eine redo-Funktionaltät, die den letzten Bearbeitungsschritt wiederholt. Die Undo- Funktion dient der Korrektur von fehlerhaften Eingaben und die Redo Funktion macht eine Korrektur rückgängig.	Should have
A17	Mehrfachselektion von Zuständen	Mehrere Zustände können durch Umrahmen mit dem Mauscursor gruppiert werden, um darauf eine Operation auszuführen wie z.B. Copy & Paste.	Should have
A18	Rastermuster	Ein Rastermuster dient der Orientierung beim Anlegen zu Zuständen	Should have
A19	Zoom	Das erstellte Zustandsdiagramm kann in seiner Ansicht vergrößert oder verkleinert werden.	Should have
A20	Neuanlegen von Masken	Von jedem Zustand aus kann per Doppelklick der integrierte GUI-Builder von Visual Studio aufgerufen werden und dem Zustand kann eine Maske generiert werden	Must have
A21	Zuordnung von Masken per Doppelklick	Wurde einem Zustand noch keine Maske zugeordnet, so werden nach einem Doppelklick auf dem Zustand alle bereits im Project definierten GUI-Masken angezeigt. Es kann eine Maske ausgewählt werden oder neu angelegt werden.	Must have

A22	Zuordnung von Masken im Property Fenster	Wurde einem Zustand noch keine Maske zugeordnet, kann im Property Fenster im Masken Property eine Referenz auf eine GUI-Klasse eingetragen werden.	Must have
A23	Ändern von Masken	Ist einem Zustand eine Maske zugeordnet, kann diese mit einem Doppelklick auf dem Zustand aufgerufen werden. Dann können Änderungen vorgenommen werden	Must have
A24	Property-Ansicht Zustand	Über diese Property-Ansicht können die Eigenschaften eines Zustands angesehen und editiert werden	Must have
A25	Property-Ansicht Startzustand	Über diese Property-Ansicht können die Eigenschaften eines Startzustandes angesehen oder editiert werden.	Must have
A26	Property-Ansicht Zustandsübergang	Über diese Property-Ansicht können die Eigenschaften eines Zustandsübergangs angesehen und editiert werden.	Must have
A27	Position Property Zustand	Über diese Property kann die Position eines Zustandes angesehen oder editiert werden.	Must have
A28	Position Property Startzustand	Über diese Property kann die Position eines Startzustandes angesehen und editiert werden.	Must have
A29	Größe Property Zustand	Über diese Property kann die Größe eines Zustandes angesehen oder editiert werden.	Must have
A30	Größe Property Startzustand	Über diese Property kann die Größe eines Anfangzustand angesehen oder editiert werden	Must have
A31	Text Property Zustand	Über diese Property kann die Beschriftung eines Zustandes angesehen oder editiert werden	Must have
A32	Text Property Startzustand	Über diese Property kann die Beschriftung eines Startzustandes angesehen oder editiert werden.	Must have
A33	Masken Property Zustand	Über diese Property kann den Eintrag (Referenz) der Maske eines Zustandes angesehen oder editiert werden.	Must have
A34	Masken Property Startzustand	Über diese Property kann den Eintrag der Maske (Referenz) eines Startzustandes angesehen oder editiert werden.	Must have
A35	Line Color Property Zustand	Über diese Property kann die Linienfarbe eines Zustandes angesehen oder editiert werden.	Nice to have

A36	Line Color Property Startzustand	Über diese Property kann die Linienfarbe eines Startzustandes angesehen oder editiert werden.	Nice to have
A37	Background Color Property Zustand	Über diese Property kann die Füllfarbe eines Zustandes angesehen oder editiert werden.	Nice to have
A38	Background Color Property Startzustand	Über diese Property kann die Füllfarbe eines Startzustandes angesehen oder editiert werden.	Nice to have
A39	Font Property Zustand	Über diese Property kann die Schriftart und Schriftgröße für einen Zustandes angesehen oder editiert werden.	Nice to have
A40	Font Property Startzustand	Über diese Property kann die Schriftart und Schriftgröße für einen Startzustand angesehen oder editiert werden.	Nice to have
A41	Notiz Property Zustand	Über diese Property kann die Notiz für einen Zustand angesehen oder editiert werden.	Should have
A42	Notiz Property Startzustand	Über diese Property kann die Notiz für einen Startzustand angesehen und editiert werden.	Should have
A43	Notiz Property Zustandsübergang	Über diese Property kann die Notiz für einen Zustandsübergang angesehen und editiert werden	Should have
A44	Event Property Zustandsübergang	Über diese Property kann das Ereignis worauf der Zustandsübergang reagieren soll, angesehen und editiert werden.	Must have
A45	Action Property	Über diese Property kann die auszuführende Aktion bei einem Zustandsübergang angesehen oder editiert werden.	Must have
A46	In Property Zustandsübergang	Der hineinführende Zustand eines Zustandsübergangs	Must have
A47	Out property	Der hinausführende Zustand eines Zustandsübergangs	Must have
A48	Points Property Zustandsübergang	Gibt eine Liste aller Knickpunkte eines Zustandsübergangs.	Should have
A49	Größe von Zuständen mit dem Mauscursor ändern	Durch das Ziehen mit dem Mauscursor an den Ecken eines Zustands kann die Größe des Zustands verändert werden.	Should have
A50	Die Position eines Zustandes über Drag & Drop ändern	Die Position eines selektierten Zustandes kann über Drag & Drop geändert werden.	Should have
A51	Die Position eines Startzustandes über Drag& Drop ändern	Die Position eines selektierten Startzustandes kann über Drag& Drop geändert werden.	Should have

A52	Die Position eines Zustandes über Drag & Drop ändern	Die Position eines selektierten Zustandes kann über Drag & Drop geändert werden.	Should have
A53	Die Position eines Zustandsübergangs über Drag & Drop ändern	Die Position eines selektierten Zustandsübergang kann über Drag & Drop geändert werden.	Should have
A54	Verschieben von Zuständen	Ein Zustand kann über Drag & Drop auf der Zeichenfläche hin und her verschoben werden.	Should have
A55	Speicherung vom Zustandsdiagramm als Bilddatei	Erstellte Zustandsdiagramme können als bmp Datei gespeichert werden	Nice to have
A56	Speicherung von Zustanddiagramm als Code	Erstellte Zustandsdiagramme können als Code gespeichert werden	Must have
A57	Änderung im Zustandsdiagrammcode (Reverse-Engeneering)	Änderung im Zustandsdiagrammcode führen zu Änderungen in Zustandsdiagramm-Editor-Ansicht	Should have
A58	Vergrößern des Zeichenblattes	Das Zeichenblatt kann bei Bedarf vergrößert werden	Should have
A60	Drucken eines Zustdandsdiagramms	Erstellte Zustandsdiagramme können seitenorientiert gedruckt werden.	Nice to have
A61	Simulieren eines Zustandsdiagramms	Zustandsdiagramme können simuliert werden.	Must have
A62	Loggen im Simulationsmodus	Alle Aktionen werden zur Übersicht ausgegeben	Should have
A63	Simulationsansicht	Es gibt eine Simulationsansicht, in der das zu simulierenden Zustandsdiagramm dargestellt wird. Hier werden alle aktiven Zustände farblich hervorgehoben.	Nice to have
A64	Scrollen	Wenn der Zeichenbereich des Zustanddiagramms größer ist als der Anzeigebereich des Monitors, erscheinen Scrollbalken, die ein Scrollen des Anzeigebereichs des aktuellen Zustandsdiagramms auf einem anderen Anzeigebereich des Zustandsdiagramms ermöglichen.	Must-have
A65	Generierung von Code	Generierung einer Fassade in c#	Must have
A66	Generierung von Code in anderen Sprachen	Generierung einer Fassade in allen .Net Sprachen	Nice to have
A67	Statechart Editor	Der Zustandsdiagramm Editor wird auf ein Statechart Editor erweitert.	Nice to have
A68	GUI-Masken-Ereignisse den Zustandsübergang zuweisen	Es muss die Möglichkeit geben, Ereignisse der GUI-Maske den Zustandsübergang zuzuweisen	Must have

Tabelle 3 Die Anforderungen an SUIT

4.3 Fazit

Es wurden die funktionalen Anforderungen an SUIT beschrieben und klassifiziert. Kriterium der Klassifikation war, ob Anforderungen konstitutiv oder optional sind, d.h. durch SUIT erfüllt werden müssen oder mögliche Ergänzungen darstellen.

5. SUIT-Architektur

5.1 Einleitung

In dem folgenden Kapitel soll die Architektur von SUIT beschrieben werden. Durch die Integration in die Entwicklungsumgebung Microsoft Visual Studio.NET sind gewisse Rahmenbedingungen für die Architektur vorgegeben. In Abschnitt 5.2 soll die architektonische Sicht beschrieben werden, wie sich SUIT in die Entwicklungsumgebung integriert. Im Abschnitt 5.3 wird dann auf den internen Aufbau von SUIT und seine Komponenten eingegangen und erläutert. Abschnitt 5.4 behandelt die Architektur des Zielprogramms, das mit Hilfe von SUIT entwickelt werden soll. Im Abschnitt 5.5 folgt dann ein Fazit.

5.2 SUIT-Architektur Zusammenhang mit VS.Net

SUIT soll sich in die Entwicklungsumgebung integrieren. Durch diese Integration in die Entwicklungsumgebung soll der Anwender die Möglichkeit haben, nicht noch extra ein weiteres zu erlernendes eigenständiges Programm zu bedienen, sondern in seiner ihm mehr oder minder wohlbekannten Entwicklungsumgebung eine Zusatzkomponente zu erhalten, die ihm in seiner Entwicklungstätigkeit hilft.

SUIT verfolgt daher den Ansatz, sich möglichst nahtlos in die Entwicklungsumgebung zu integrieren und sich in der Bedienung und Handhabung ähnlich wie die Entwicklungsumgebung selbst zu verhalten.

Diese Integration ist jedoch nur möglich, wenn die Entwicklungsumgebung eine Integration erlaubt, sie durch geeignete Schnittstellen unterstützt.

Microsofts Visual Studio.NET bietet für die Integration das Automatisierungsmodell EnvDTE und die Design-Time-Architektur in .NET erlaubt es, auf Designerfunktionalitäten – wie z.B. die GUI-Builder-Funktionalitäten von Visual Studio.NET – zuzugreifen (siehe Kapitel 6).

Durch die Integration von SUIT in Visual Studio.NET ist die Möglichkeit gegeben, über diese Schnittstellen auf die Entwicklungsumgebung selbst, bzw. auf die in Visual Studio.NET integrierten GUI-Builder zuzugreifen. Somit muss eine wichtige Komponente für SUIT nicht mehr selbst entwickeln werden.

Wie sich SUIT nun in die bestehende .NET-Architektur integriert zeigt Abb. 23.

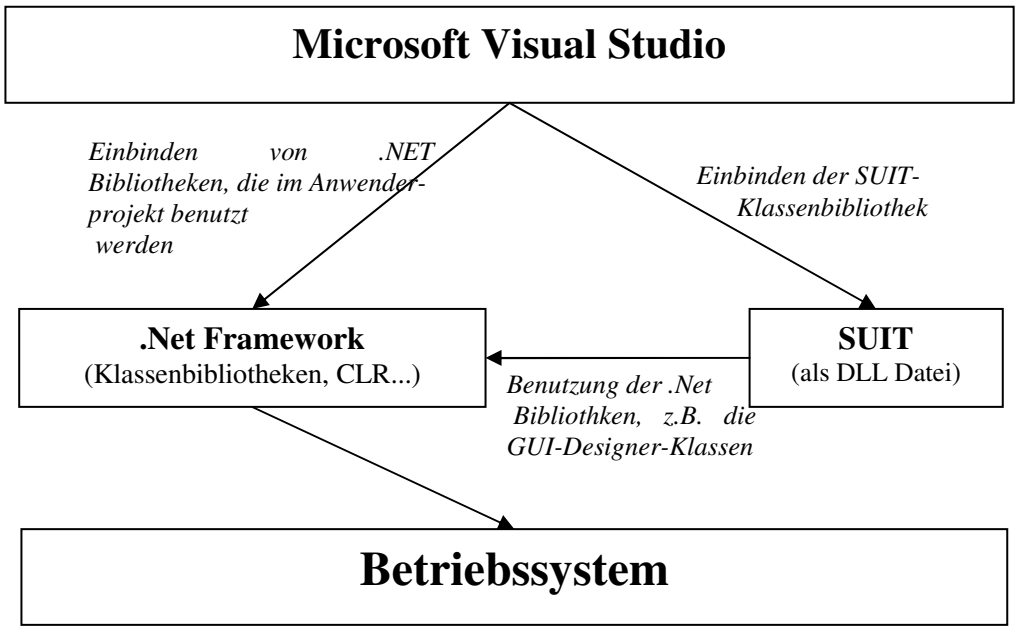


Abb. 23 Integration von SUIT in die Microsoft Visual Studio.Net Architektur

Die Architektur in Abb. 23 zeigt eine stark vereinfachte Darstellung des Zusammenspiels von Visual Studio und dem .Net-Framework. Aus der Abbildung lässt sich entnehmen, dass SUIT sich auf gleicher Stufe wie das .Net Framework ansiedelt, jedoch die .Net Framework Klassenbibliothek mitbenutzt.

In Microsoft Visual Studio lassen sich Klassenbibliotheken aus dem .Net Framework in den Anwendercode einbinden. SUIT verhält sich hier genauso wie eine .Net Klassenbibliothek, die der Anwender auch über Visual Studio in seinem Projekt einbinden kann. Somit ist die Handhabung von SUIT für den Anwender nicht anders, als er es schon aus den .Net Klassenbibliotheken gewohnt war.

Die Kapselung des Betriebssystems vom Anwendercode, also von den Quellcodes, die über Visual Studio geschrieben und verwaltet werden, geschieht weiterhin über das .Net Framework. SUIT integriert sich hier nahtlos, und durchbricht diese Kapselung nicht.

5.3 Interne SUIT-Architektur

In diesem Abschnitt soll auf die wesentlichen Bestandteile von SUIT eingegangen werden. SUIT lässt sich auf folgenden 4 Komponenten aufteilen:

- auf die Zustandsdiagramm-Komponente
- auf die GUI-Builder-Komponente
- auf die Simulationskomponente
- auf die Codeerzeugungskomponente

Auf die Entwicklung eines neuen GUI-Builders wurde verzichtet, da der GUI-Builder von Visual Studio sich hier anbietet, wiederbenutzt zu werden und dadurch die Realisierung eines eigenen GUI-Builders nicht erforderlich ist. Somit kann Entwicklungszeit gespart werden,

und der Anwender bekommt ein GUI-Builder vorgesetzt, mit dem er mehr oder minder vertraut ist. Es wird somit auf die GUI-Builder-Komponente zurückgegriffen, die Visual-Studio bereits zur Verfügung stellt. Dieser Zugriff ist dank des Service-Hierarchy Konzept von .Net möglich, da in .Net mit diesem Service auch auf Instanzen des GUI-Builder Designer zugegriffen werden kann.

Durch die Wiederverwendung des bereits in Visual Studios bestehenden GUI-Builders lässt sich SUIT letztendlich in 3 eigenen Komponenten und eine fremd entwickelte Komponente aufteilen.

Abb. 24 zeigt das Zusammenspiel dieser Komponenten.

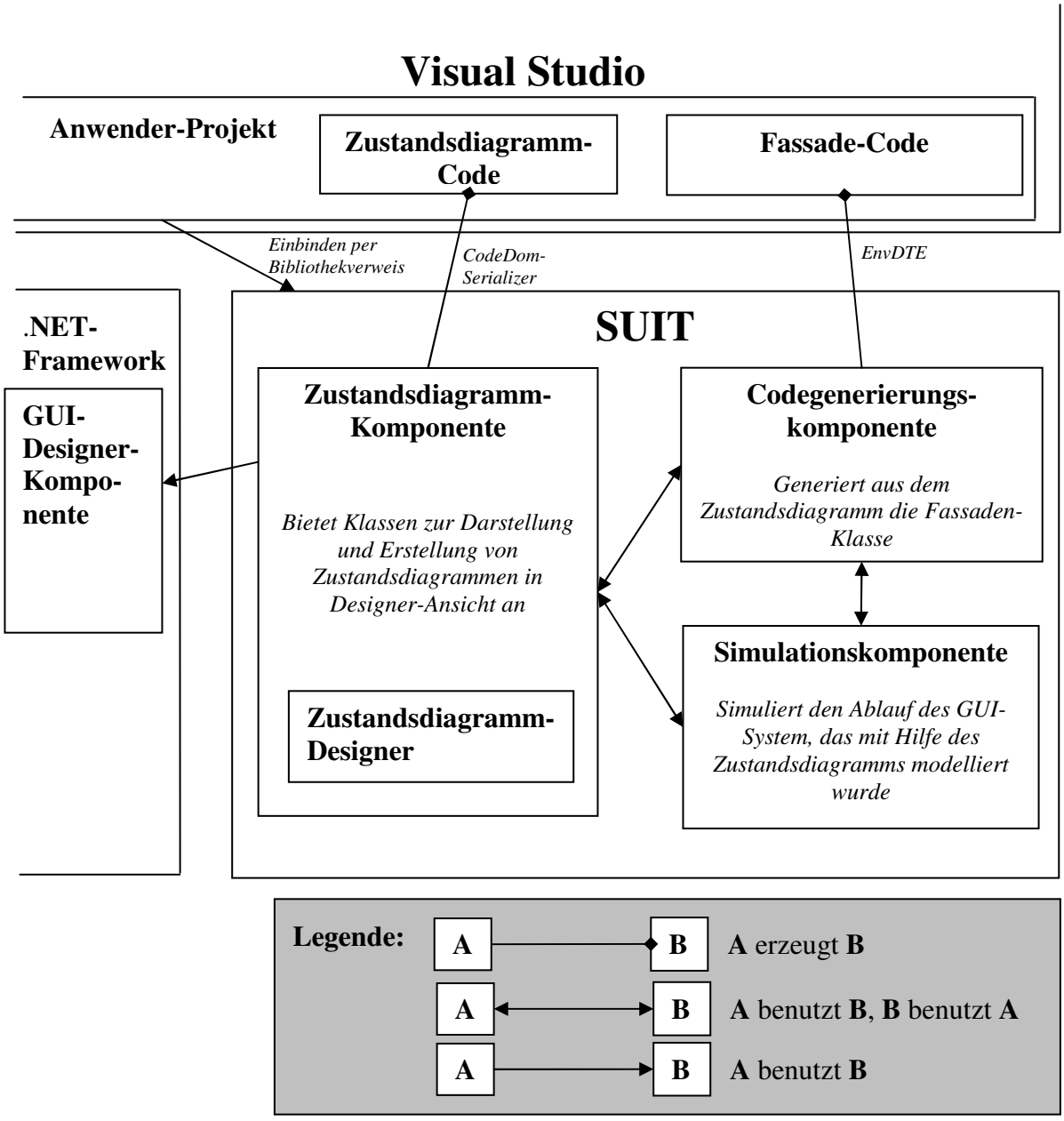


Abb. 24 Zusammenspiel der SUIT Komponenten

In Abb. 24 erkennt man, dass SUIT sich aus 3 Komponenten, der Zustandsdiagramm-Komponente, Codegenerierungskomponente und der Simulationskomponente zusammensetzt. Die GUI-Designer-Komponente des .Net-Frameworks wird von der Zustandskomponente immer dann mitbenutzt, wenn der Anwender aus seinem Zustandsdiagramm heraus den GUI-Builder aufrufen will. Dies geschieht zum Beispiel dann, wenn der Anwender einem Zustand eine GUI-Maske zuordnen will.

Durch das Einbinden von SUIT über Bibliothekenverweise an das Anwender-Projekt bekommt der Anwender die Möglichkeit, Zustandsdiagramme in Anwendungscode zu erzeugen. Dieser Code, der in einer beliebigen .Net Sprache geschrieben sein kann, wird durch die Zustandsdiagramm-Komponente als grafisches Zustandsdiagramm umgesetzt und im Zustandsdiagramm-Designer angezeigt. Dieses .Net sprachübergreifende Einlesen und Speichern von Code ist dank dem CodeDomSerializer möglich, der in .Net für jedes Objekt, dass das IComponent Interface implementiert, ein CodeDom Objekt erzeugen und dieses dann serialisieren kann. Ein CodeDom Objekt ist ein Objekt Model, das Quellcode repräsentiert. Es sprachunabhängig konzipiert (vergleiche Kapitel 6).

Der Zustandsdiagramm-Designer ist Bestandteil der Zustandsdiagramm-Komponente und bietet dem Anwender die Möglichkeit, per Drag&Drop Verfahren und Properties-Einstellungen sein Zustandsdiagramm aufzubauen, zu editieren und zu entwickeln. Dabei wurde das Ziel gesetzt, dass der Zustandsdiagramm-Designer sich möglichst nahtlos in die .Net Entwicklungsumgebung integriert (siehe Abschnitt 5.2). So wurde die schon aus dem GUI-Builder bekannte Toolbox beim Zustandsdiagramm-Designer um neue Elemente erweitert, wie z.B. Zustand und Zustandsübergang. Hierbei wird das Ziel verfolgt, dass SUIT sich mit wenig Aufwand weiterentwickeln lässt. Im Falle einer Weiterentwicklung kann die Toolbox ohne einen großen Aufwand um weitere Zustandsdiagrammelemente ergänzt werden.

Um diese Ergänzung zu ermöglichen, wurde eine abstrakte Oberklasse ZustandsdiagrammElement angelegt, von der alle Toolbox Elemente für den Zustandsdiagramm-Designer erben muss. Der Zustandseditor von SUIT ist so angelegt, dass er alle Elemente von Typ ZustandsdiagrammElement akzeptiert. Will man also den SUIT Zustandsdiagrammeditor um weitere Elemente erweitern, so muss man eine Klasse definieren, die von ZustandsdiagrammElement erbt. Abb. 25 zeigt die Vererbungsstruktur der Elemente, die zur Zeit in SUIT unterstützt werden.

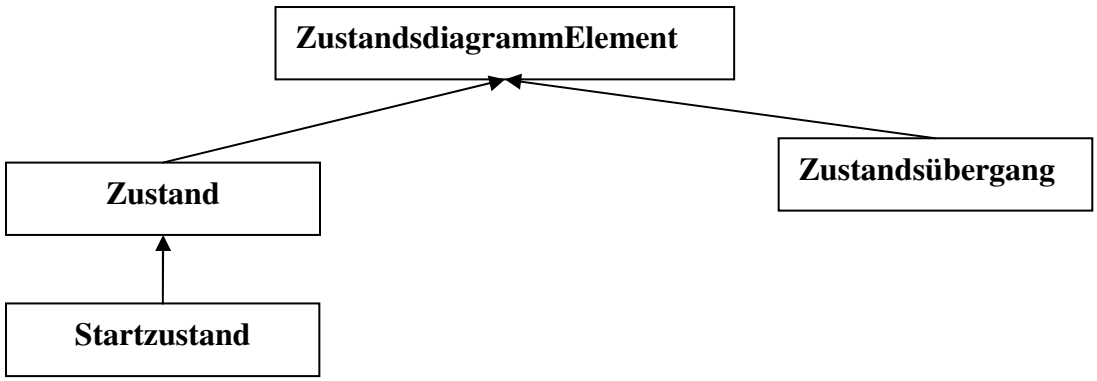


Abb. 25 Vererbungsbaum der Elemente in SUIT Zustandsdiagramm-Designer

In Abb. 25 lässt sich erkennen, dass alle zur Zeit im SUIT unterstützten Elemente, Zustand, Startzustand und Zustandübergang direkt oder indirekt von ZustandsdiagrammElement erben. Durch diese Vererbung werden diese Elemente vom SUIT Zustandsdiagrammeditor als erlaubte Toolbox-Elemente erkannt. Wie dies vonstatten geht, wird in Kapitel 6 erklärt.

Das Property-Fenster, das auch beim mitgelieferten GUI-Builder von Visual Studio.Net bekannt ist, wird auch beim Zustandsdiagramm-Designer erneut eingeführt, um dem Anwender eine vertraute Entwicklungsumgebung anzubieten.

Über den Zustandsdiagramm-Designer lässt sich dann die Codegenerierungskomponente aufrufen. Die Codegenerierungskomponente generiert anhand des Zustandsdiagramms, das vom Zustandsdiagrammdesigner entworfen wurde, eine Fassade-Klasse, die die Methodenköpfe der im Zustandsdiagramm vorkommenden Aktionen definiert. Dies sollte ursprünglich über CodeDom geschehen, jedoch stellte sich hier heraus, dass das CodeDom Modell in der vorliegenden .Net Framework Version 1.1 für Objekte, die nicht von Typ IComponent sind, noch nicht komplette Sprachunabhängigkeit erreicht hat. In Kapitel 6 wird auf CodeDom näher eingegangen.

Die Simulationskomponente wird auch vom Zustandsdiagramm-Designer angestoßen. Sie nutzt die Codegenierungskomponente, um die Fassadeklasse zu genieren, sofern dies nicht schon vom Anwender eingeleitet wurde. Wenn Fassadeklasse und die GUI-Masken entworfen sind, kann die Codegenierungskomponente anhand des Zustandsdiagramms die GUI-Masken kompilieren und nach der im Zustandsdiagramm beschriebenen Ablauflogik ausführen.

5.4 Architektur des Zielsystems

Durch die Codegenerierungskomponente von SUIT wird für den Anwender Code erzeugt. Jedoch beschränkt sich dieser automatisch erzeugte Code nur auf eine Fassaden-Klasse, die die Methodenköpfe der im Zustandsdiagramm angebenden Aktionen deklariert. Die Ausimplementierung – also die eigentliche Funktionalität - kann nur vom Anwender selbst entwickelt und programmiert werden. Eine automatische Codegenerierung für die dahinter liegende Funktionalität ist nicht möglich, da die nötigen Informationen hierzu nicht aus dem Zustandsdiagramm entnommen werden können.

Durch die Generierung einer Fassaden-Klasse wird dem Anwender also schon ein gewisses Entwurfsmuster, ein Pattern, vorgeben, das somit auch eine gewisse Entwurfsentscheidung für den Anwender darstellt. Es entsteht eine Vorgabe in der Architektur des Zielprogramms, das vom Anwender mit Hilfe von SUIT entwickelt werden soll. Diese Architektur soll in diesem Abschnitt vorgestellt und deren Vorteile erläutert werden.

Das Fassaden-Pattern [Gamma 2004] hat den Vorteil, dass es den Klient – im unserem Fall die Interaktionskomponente – vom Subsystem, die Funktionskomponente, voneinander entkoppelt, und eine starke Kopplung somit vermieden werden kann.

Die Interaktionskomponente, also die Komponente, die die GUI enthält, wird mit Hilfe von SUIT und dem in SUIT integrierten GUI-Builder entworfen. Durch die automatisch generierte Fassaden-Klasse wird die Schnittstelle angeboten, die diese Interaktionskomponente mit der Funktionalen Komponente kommunizieren lässt. Abb. 26 zeigt die dadurch entstehende Architektur des Anwendungssystems.

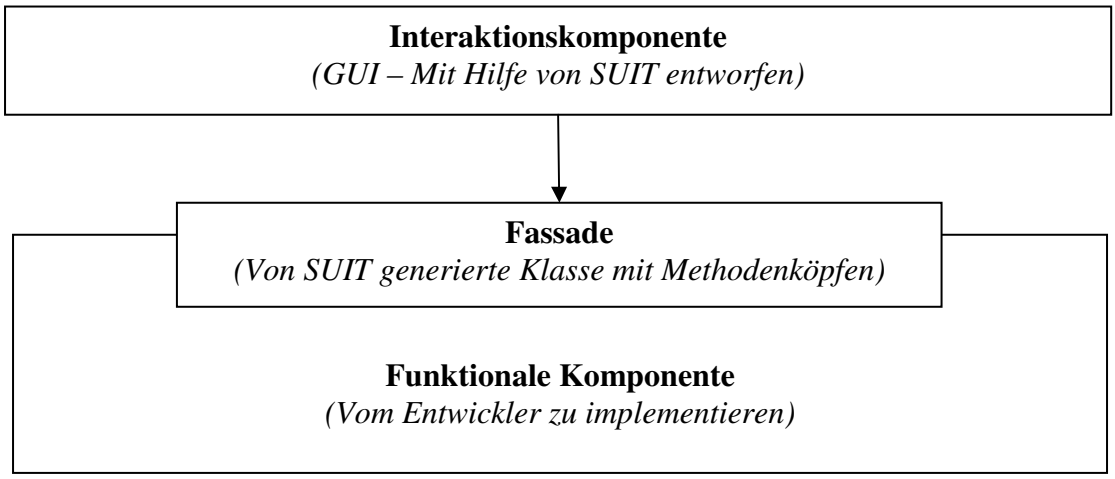


Abb. 26 Architektur des Zielsystem mit Fassade

Die Interaktionskomponente wird durch SUIT dahingehend angereichert, dass die Aktionen, die im SUIT-Zustandsdiagramm angegeben werden, auch aufgerufen werden, wenn das zur Aktion zugehörige Ereignis ausgelöst wird. Dieser Methodenaufruf ruft die Methodendeklaration in der Fassaden-Klasse auf, von der der Anwender durch nachträgliches Ausimplementieren auf die entsprechenden Klassen der Funktionalen Komponente delegieren kann.

So ist die Interaktionskomponente nur an die Fassade gekoppelt. Der Entwurf der funktionalen Komponente ist dem Anwender überlassen, sodass trotz einer von SUIT vorgegebenen Zielsystem-Architektur genügend Freiheiten in Bezug auf die die Gestaltung seines Systems gegeben ist.

Auch ist die Gestaltung der Interaktionskomponente – die Gestaltung der GUI – dem Anwender frei überlassen, und SUIT bietet sich hier nur als unterstützendes Tool für Gestaltung und Entwurf der Interaktionskomponente an. Die Auswertung der Benutzereingaben und die Präsentation der Ausgaben (die Ver- und Entsorgung der Felder) werden in SUIT nicht berücksichtigt, um diese muss sich der Anwender selbst kümmern (siehe Abschnitt 3.4).

Dadurch ist dem Anwender – trotz vorgegebenen Fassade-Pattern – genügend Freiraum gelassen, um seinen eigene Designentwurfe einzubringen. Dennoch sorgt diese durch die Fassade erzwungene Entkoppelung dafür, dass eine saubere Trennung von Funktionalität und View stattfinden muss, sodass der Anwender mehr oder minder verpflichtet ist, auf Softwareengeneeringssicht ein gutes Entwurfskonzept zu realisieren.

5.5 Fazit

Die SUIT-Architektur gliedert sich in die bestehende Visual Studio.Net Architektur ein. SUIT verhält sich hierbei in der Handhabung wie eine .Net Klassenbibliothek. Die interne SUIT Architektur lässt sich in drei eigen entwickelte Komponenten (Zustandsdiagramm-

Komponente, Simulationskomponente und Codegenierungskomponente) und eine fremdentwickelte Komponente (GUI-Designer-Komponete) unterteilen.

Durch die automatische Fassaden-Generierung wird eine Schnittstelle zu sauberen Trennung zwischen der Interaktionskomponente und der Funktionalen Komponente angeboten, die der SUIT-Anwender bei dem Entwurf seines Zielsystem-Architektur berücksichtigen muss.

6. Umsetzung von SUIT

6.1 Einleitung

Dieses Kapitel gibt einen kleinen Einblick in die Umsetzung von SUIT. In Abschnitt 6.2 wird darauf eingegangen, welche der Anforderung im Anforderungskapitel (siehe Kapitel 4) umgesetzt wurden sind. Im Abschnitt 6.3 wird die Programmstruktur von SUIT erläutert. In Abschnitt 6.4 werden die Besonderheiten der Umsetzung von SUIT und die für die Umsetzung des Tools verwendeten .Net Technologien beschrieben. In Abschnitt 6.5 wird die Schnittstelle zu SUIT erläutert. In 6.6 folgt ein Fazit.

6.2 Umgesetzte Anforderungen

Die im Kapitel 4 beschriebenen Anforderungen werden in diesem Abschnitt aufgegriffen. Es soll hier erläutert werden, welche dieser Anforderung mit der aktuellen SUIT Version realisiert wurden und welche davon noch offen sind. Im Folgenden wird eine Tabelle dargestellt, die den Status (erfüllt / nicht erfüllt) der einzelnen Anforderungen wiedergibt:

ID	Abkürzung	Status	Klassifikation
A1	Erzeugen einer Zustandsdiagrammklasse	Erfüllt	Must have
A2	Toolbox	Erfüllt	Must have
A3	Erzeugen eines Zustandes per Drag & Drop	Erfüllt	Must have
A4	Erzeugen eines Anfangszustandes per Drag & Drop	Erfüllt	Must have
A5	Erzeugen eines Zustandes per Doppelklick	Erfüllt	Should have
A6	Erzeugen eines Startzustand per Doppelklick	Erfüllt	Should have
A7	Erzeugen eines Zustandsübergang (Transition) per Drag&Drop	Erfüllt	Must have
A8	Erzeugen eines rekursiven Zustandsübergangs	Erfüllt	Should have
A9	Löschen eines Zustandes	Erfüllt	Must-have
A10	Löschen eines Startzustandes	Erfüllt	Must Have
A11	Löschen eines Zustandsübergangs	Erfüllt	Must have
A12	Zustandsübergänge Knickpunkte	Erfüllt	Should have
A13	Notizfeld	nicht erfüllt	Nice to have
A14	Selektieren	Erfüllt	Must have
A15	Selektieren des gesamten Zustandsdiagramms	nicht erfüllt	Should have
A16	Undo/ redo	Erfüllt	Should have
A17	Mehrfachselektion von Zuständen	Erfüllt	Should have
A18	Rastermuster	Erfüllt	Should have
A19	Zoom	Erfüllt	Should have
A20	Neuanlegen von Masken	Erfüllt	Must have

A21	Zuordnung von Masken per Doppelklick	Erfüllt	Must have
A22	Zuordnung von Masken im Property Fenster	Erfüllt	Must have
A23	Ändern von Masken	Erfüllt	Must have
A24	Property-Ansicht Zustand	Erfüllt	Must have
A25	Property-Ansicht Startzustand	Erfüllt	Must have
A26	Property-Ansicht Zustandsübergang	Erfüllt	Must have
A27	Position Property Zustand	Erfüllt	Must have
A28	Position Property Startzustand	Erfüllt	Must have
A29	Größe Property Zustand	Erfüllt	Must have
A30	Größe Property Startzustand	Erfüllt	Must have
A31	Text Property Zustand	Erfüllt	Must have
A32	Text Property Startzustand	Erfüllt	Must have
A33	Masken Property Zustand	Erfüllt	Must have
A34	Masken Property Startzustand	Erfüllt	Must have
A35	Line Color Property Zustand	Erfüllt	Nice to have
A36	Line Color Property Startzustand	Erfüllt	Nice to have
A37	Background Color Property Zustand	Erfüllt	Nice to have
A38	Background Color Property Startzustand	Erfüllt	Nice to have
A39	Font Property Zustand	Erfüllt	Nice to have
A40	Font Property Startzustand	Erfüllt	Nice to have
A41	Notiz Property Zustand	Erfüllt	Should have
A42	Notiz Property Startzustand	Erfüllt	Should have
A43	Notiz Property Zustandsübergang	Erfüllt	Should have
A44	Event Property Zustandsübergang	Erfüllt	Must have
A45	Action Property	Erfüllt	Must have
A46	In Property Zustandsübergang	Erfüllt	Must have
A47	Out property	Erfüllt	Must have
A48	Points Property Zustandsübergang	Erfüllt	Should have
A49	Größe von Zuständen mit dem Mauscursor ändern	Erfüllt	Should have
A50	Die Position eines Zustandes über Drag & Drop ändern	Erfüllt	Should have
A51	Die Position eines Startzustandes über Drag& Drop ändern	Erfüllt	Should have
A52	Die Position eines Zustandes über Drag & Drop ändern	Erfüllt	Should have
A53	Die Position eines Zustandsübergangs über Drag & Drop ändern	Erfüllt	Should have
A54	Verschieben von Zuständen	Erfüllt	Should have
A55	Speicherung vom Zustandsdiagramm als Bilddatei	Erfüllt	Nice to have
A56	Speicherung von Zustanddiagramm als Code	Erfüllt	Must have
A57	Änderung im Zustandsdiagrammcode (Reverse-Engeenering)	Erfüllt	Should have
A58	Vergrößern des Zeichenblattes	Erfüllt	Should have
A60	Drucken eines Zustdandsdiagramms	nicht erfüllt	Nice to have
A61	Simulieren eines Zustandsdiagramms	Erfüllt	Must have
A62	Loggen im Simulationsmodus	Erfüllt	Should have
A63	Simulationsansicht	Erfüllt	Nice to have
A64	Scrollen	Erfüllt	Must-have

A65	Generierung von Code	Erfüllt	Must have
A66	Generierung von Code in anderen Sprachen	nicht erfüllt	Nice to have
A67	Statechart Editor	nicht erfüllt	Nice to have
A68	GUI-Masken-Ereignisse den Zustandsübergang zuweisen	Erfüllt	Must have

Tabelle 4 Umgesetzte Anforderungen

6.3 Programmstruktur

Im Folgenden wird ein Ausschnitt aus dem Klassendiagramm von SUIT dargestellt. Das vollständige Klassendiagramm befindet sich im Anhang.

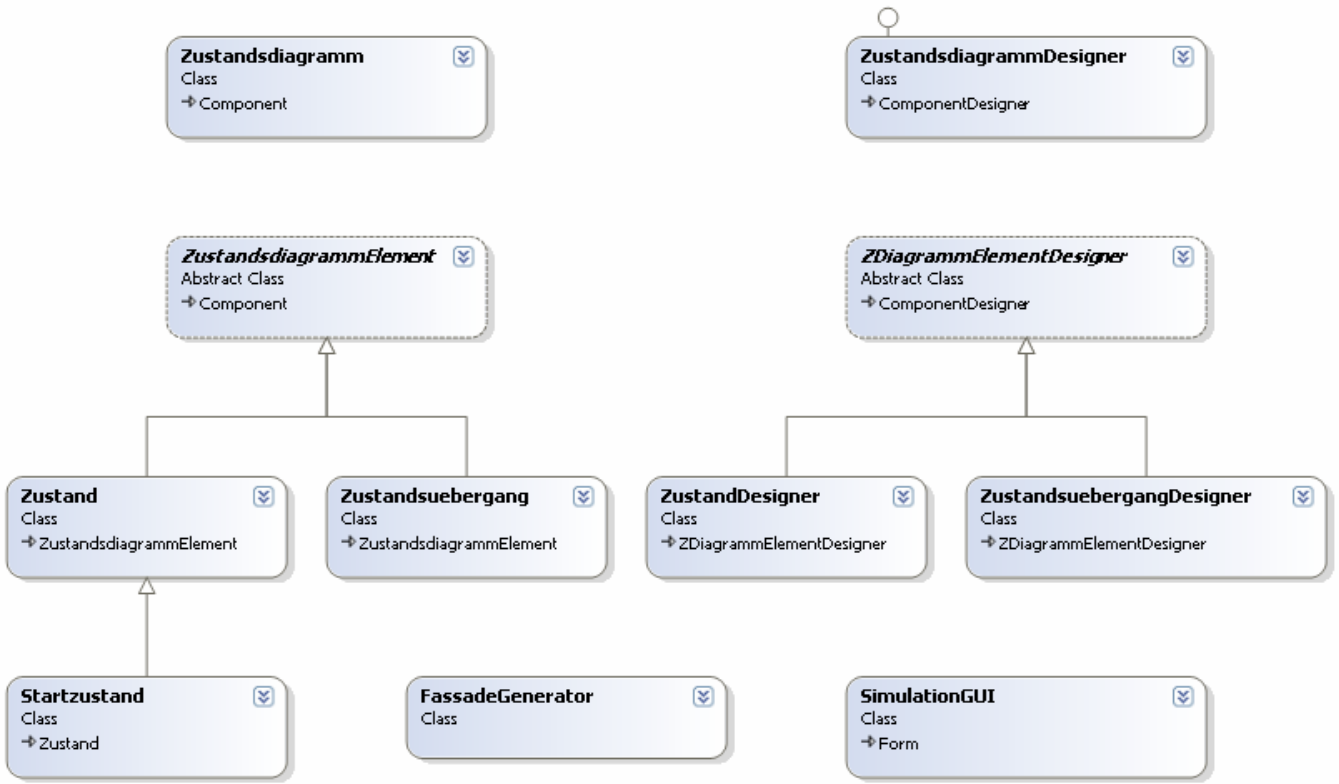


Abb. 27 Ausschnitt aus dem Klassendiagramm

Abb. 27 zeigt einen Ausschnitt aus dem Klassendiagramm von SUIT. In diesem Ausschnitt sind bis auf die Vererbungsbeziehungen keine weiteren Beziehungen in der Abbildung dargestellt. Dies lässt sich auf die Benutzung von Attribut-Klassen in .Net begründen. Die Attribute-Klassen ordnen einer Klasse vordefinierte Systeminformationen oder benutzerdefinierte Informationen zu (siehe dazu [msdn-attribute]). Es handelt sich hierbei um Definitionen, die in einer Klasse zwischen zwei eckigen Klammern beschrieben werden.

Die Klassem stehen also durch die Definition der Attribute-Klassen miteinander in Beziehung. Die in UML bekannten Beziehungen wie Aggregationen oder Kompositionen kann hier nicht immer definiert werden, da die Beziehungen durch die Attribute-Klassen zustande kommen.

Die Klassen in SUIT lassen sich, wie anhand des Klassendiagramms (Abb. 27) erkennbar, grob in Designer- und Zustandsdiagrammelemente einteilen: Die Designer Klassen und die Komponentenklassen der Zustandsdiagrammelemente und des Zustandsdiagramms.

Jede Designer-Klasse ist eine Klasse vom Typ **ZustandsdiagrammElement** oder **Zustandsdiagramm** zugeordnet. Die Designerklasse **ZustandsdiagrammDesigner** z.B. ist der Klasse **Zustandsdiagramm** zugeordnet.

Die Designerklassen erweitern das Verhalten der Komponenten Zustandsdiagrammelemente und Zustandsdiagramm zur Entwurfszeit. Sie bieten Operationen zum Löschen, Erzeugen und Editieren dieser Komponenten. Die Komponentenklassen repräsentieren dagegen die Daten der einzelnen Zustandsdiagrammkomponenten.

ZustandsdiagrammElement ist eine abstrakte Klasse, von **Component** abgeleitet. Sie stellt eine Zustandsdiagrammelement-Komponente dar. **ZustandsdiagrammElement** stellt Methoden zum Zeichnen, zum Löschen und zur Initialisierung von Zustandsdiagramm-Elementen bereit.

Zustandsdiagramm ist von **Component** abgeleitet. Diese Klasse stellt ein Zustandsdiagramm dar. Die Klasse **Zustandsdiagramm** verwaltet eine Liste von Zustandsdiagrammelementen.

Zustand ist eine Unterklasse der Klasse **ZustandsdiagrammElement**. Sie stellt einen Zustand dar. Sie implementiert die abstrakten Methoden von **ZustandsdiagrammElement.**

Startzustand erweitet die Klasse **Zustand** um eine Startzustandkomponente.

Zustandsuebergang ist von **ZustandsdiagrammElement** abgeleitet und stellt einen Zustandsübergang dar. In **Zustandsuebergang** werden die abstrakten Methoden von **ZustandsdiagrammElement** implementiert.

ZustandsdiagrammDesigner stellt einen Designer bereit, der das Verhalten eines Zustandsdiagramms im Entwurfsmodus erweitert. Der **ZustandsdiagrammDesigner** ist Bestandteil der Zustandsdiagramm-Komponente und bietet dem Anwender die Möglichkeit, per Drag&Drop-Verfahren und über Properties-Einstellungen sein Zustandsdiagramm aufzubauen, zu editieren und zu entwickeln. Vom ZustandsdiagrammDesigner aus kann die Fassadengenerierung und die Simulation angestoßen werden.

ZDiagrammElementDesigner ist eine abstrakte Klasse. Sie stellt einen Designer bereit, der das Verhalten einer Komponente vom Typ **ZustandsdiagrammElement** im Entwurfsmodus erweitert. Sie bietet Methoden zum Ermitteln der Mausposition auf einem Zustandsdiagrammelement und stellt Methoden zum Zeichnen bereit.

ZustandDesigner ist eine von **ZDiagrammElementDesigner** abgeleitete Klasse. Die Klasse **ZutandsDesigner** erweitert das Verhalten eines Zustands im Entwurfsmodus.

ZustandsuebergangDesigner ist eine von **ZDiagrammElementDesigner** abgeleitete Klasse. Sie erweitert das Verhalten eines Zustandsübergangs im Entwurfsmodus.

FassadeGenerator ist eine Klasse, in der die Fassade des erzeugten Zustandsdiagramms generiert wird.

6.4 Besonderheiten

6.4.1 Erweiterung der Toolbox um Zustandsdiagramm-Elemente

Die Toolbox besteht in der Ansicht des GUI-Builders in Visual Studio aus einer großen Anzahl von Toolboxelementen wie Buttons, TextBoxes, Listen etc. Diese Toolbox ist in jedem Designer verfügbar. Abhängig von dem zurzeit verwendeten Designer ist es nicht immer sinnvoll, in der Toolbox immer dieselben Toolboxelemente zur Verfügung zu stellen. So sind z.B. Buttons, Listen und andere für den GUI-Builder zur Verfügung stehende Toolboxelemente für den ZustandsdiagrammDesigner uninteressant.

Stattdessen ergibt es Sinn, für einen ZustandsdiagrammDesigner in der Toolbox Elemente wie z.B. einen Zustand, einen Zustandsübergang, einen Startzustand etc. anzubieten und alle anderen Toolboxelemente zu deaktivieren. Um dies zu erreichen, benutzten wir einen Filter, der es erlaubt, nur bestimmte Toolboxelemente in der Toolbox zu aktivieren.

Damit Objekte der von **ZustandsdiagrammElement** erbenden Komponenten **Startzustand**, **Zustand** und **Zustandsuebergang** in der Toolbox für den ZustandsdiagrammDesigner aktiviert werden können, müssen die von **System.ComponentModel.Component** abgeleitete Klasse **ZustandsdiagrammElement** und die Designer-Klasse **ZustandsdiagrammDesigner** vor der Klassendeklaration folgenden Code enthalten:

```
[ToolboxItemFilter("SUIT.ZustandsdiagrammEditor.ZustandsdiagrammElement",
ToolboxItemFilterType.Require)]
```

ToolboxItemFilter(String, ToolboxItemFilterType) gibt die für ein Toolboxelement zu verwendende Filterzeichenfolge und den Filtertyp an.

Der *ToolboxItemFilterType ToolboxItemFilterType.Require***,** gibt an, dass die Filterzeichenfolge "*SUIT.ZustandsdiagrammEditor.ZustandsdiagrammElement*" für ein Toolboxelement vorhanden sein muss, um ein Toolboxelement zu aktivieren. Diese Bedingung ist für den **Startzustand**, **Zustand** und **Zustandsübergang** gegeben, da diese Klassen von **ZustandsdiagrammElement** abgeleitet sind und somit die Filterzeichenfolge "*SUIT.ZustandsdiagrammEditor.ZustandsdiagrammElement*" erben.

Des Weiteren müssen die Klassen **ZustandsdiagrammDesigner** und **ZustandsdiagrammElement** über eine übereinstimmende Filterzeichenfolge verfügen. In unserem Code verfügen beide Klassen über die Filterzeichenfolge "*SUIT.ZustandsdiagrammEditor.ZustandsdiagrammElement*".

Die Toolboxitems vom Typ **Startzustand**, **Zustand** und **Zustandsübergang** erfüllen jetzt die Bedingung innerhalb des Zustandsdiagrammdesigners referenziert und benutzt zu werden. Sie erscheinen aber noch nicht auf der Toolbox (Abb. 28), da sie dieser noch nicht hinzugefügt wurden:

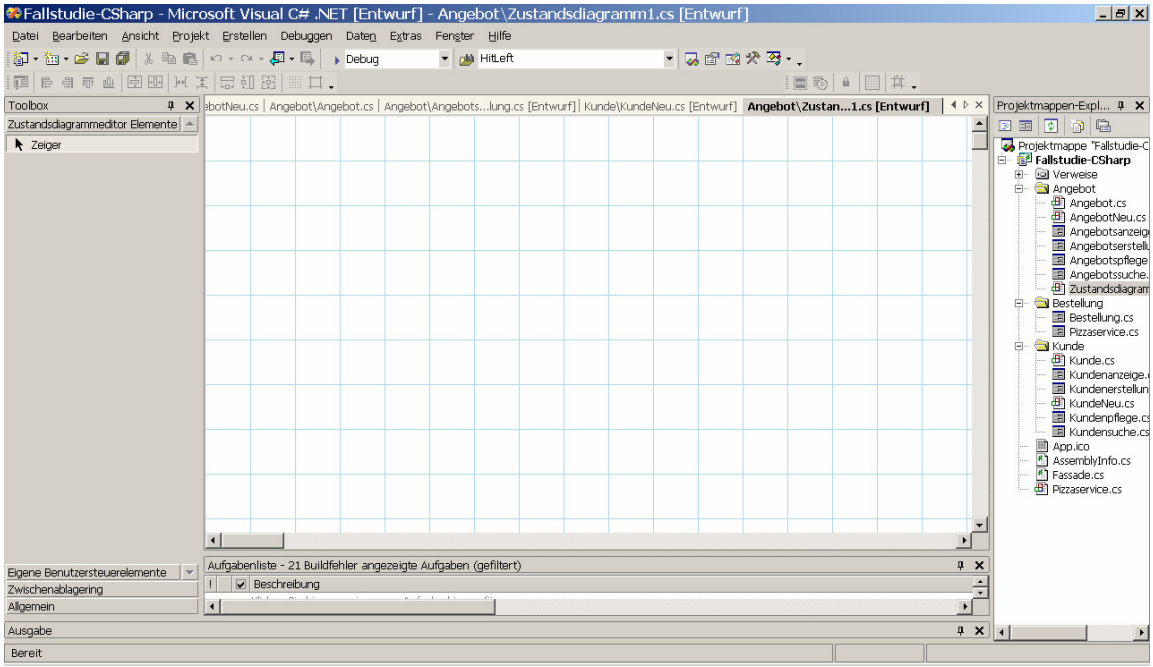


Abb. 28 Toolbox vor dem Hinzufügen der Toolboxitems für das Zustandsdiagramm

Die Toolboxelemente müssen nun in der Toolbox platziert werden, damit sie in ihr erscheinen. Das Hinzufügen der bereits definierten Komponenten kann manuell über die Visual Studio-Oberfläche erfolgen. Mit einem rechten Mausklick auf der Toolbox können diese Elemente über die Option ‚Elementen hinzufügen/entfernen' ausgewählt und hinzugefügt werden. Siehe dazu Abb. 29.

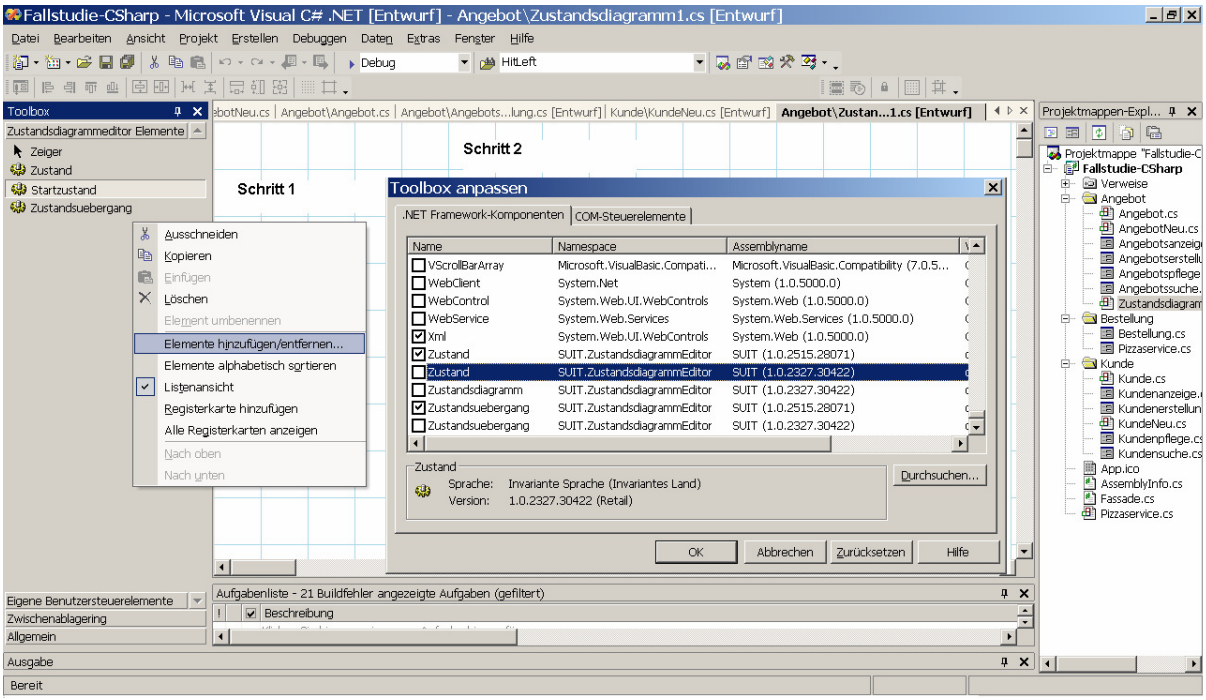


Abb. 29 Toolbox nach dem Hinzufügen der Toolboxitems für das Zustandsdiagramm

Um den Vorgang des Einfügens der Komponenten in die Toolbox zu automatisieren, implementiert die Designer-Klasse **ZustandsdiagrammDesigner** die Methode *addToolBoxItems*; auf diese Weise muss dieser Vorgang nicht mehr manuell durchgeführt werden. Wird eine Klasse vom Typ **Zustandsdiagramm** mit dem

ZustandsdiagrammDesigner geöffnet, so fügt dieser der Toolbox ein Startzustand-, ein Zustand- und ein Zustandsübergang-Item automatisch hinzu.

6.4.2 Öffnen mit dem Zustandsdiagramm-Designer in VS.Net

Will man in Visual Studio eine Klasse in der Designeransicht öffnen, so kann dies mit einem Doppelklick auf die Klasse erreicht werden. Das Anzeigen einer Klasse in der Designeransicht ist nur möglich, wenn eine Klasse bereits einem Designer zugeordnet worden ist. In Visual Studio werden Klassen, die von **System.Windows.Forms.Form** abgeleitet sind, in der Designeransicht des GUI-Builders angezeigt.

Form ist von **Component** abgeleitet. In .Net können Komponentenklassen Designerklassen zugeordnet werden. Form ist dem Designer GUI-Builder zugeordnet. Darin liegt begründet, dass beim Öffnen einer Klasse vom Typ Form diese in der Designeransicht des GUI-Builders geöffnet wird.
In SUIT soll das Zustandsdiagramm in der Zustandsdiagrammdesigner-Ansicht geöffnet werden. Es muss also der Komponentenklasse Zustandsdiagramm die Designerklasse **Zustandsdigrammdesigner** zugeordnet werden. Dieser Vorgang wird im Folgenden angezeigt. Der **Zustandsdigrammdesigner** ist der Klasse **Zustandsdiagramm** durch folgende Code-Definition zugeordnet:

```
[Designer(typeof(SUIT.ZustandsdiagrammEditor.Design.ZustandsdiagrammDe-
signer), typeof(IRootDesigner))]
```

Designer (Type DesignerTypeName, Type DesignerBaseType) wird vor der Klassendeklaration aufgerufen. Dieser Code initialisiert eine neue Instanz der **DesignerAttribute**-Klasse mit den Typen des Designers und der Designerbasisklasse.

Mit dem Parameter Type *DesignerTypeName* wird der zuzuordnende Designer übergeben. In unserem Code ist es der **ZustandsdiagrammDesigner**.

typeof(IRootDesigner) sagt aus, dass der ZustandsdiagrammDesigner vom Typ IRootDesigner ist. Ein Designer vom Typ IRootDesigner wird beim Doppelklick auf die Klasse angezeigt. Er ist der Stammdesigner und befindet sich an der höchsten Position.

6.4.3 Programmiersprachunabhängigkeit in der Codegenerierung in .Net

Verwenden von CodeDom

.NET stellt den **Code Document Object Model (CodeDom)-** Mechanismus zur Verfügung, um die logische Struktur des Quellcodes zu beschreiben. **CodeDom-Elemente** werden miteinander verbunden, sodass eine Datenstruktur entsteht, die die Struktur von Quellcode abbildet. Jede Klasse, Methode, Variable etc. ist ein eigenes Objekt, das instanziert und zum Diagramm hinzugefügt werden muss. Im Folgenden wird an einem Beispielcode aus [CodeDom] die Erzeugung eines **CodeDom** Diagramm demonstriert.

Beispiel:

```
//In c#
 Point myPoint = new Point(10, 10);

// in VB
Dim myPoint As Point = New Point(10, 10)
```

Im Beispielcode wird ein Objekt der Klasse Point mit seinen x- und y-Koordinaten erzeugt. Aus diesem Programmiercode wird nun demonstriert, wie eine CodeDom Struktur aufgebaut werden kann.

Als erstes wird eine Variablendeklaration vom Typ **Point** mit dem Namen „myPoint“ erzeugt.

```
CodeVariableDeclarationStatement myPointVariable;
myPointVariable = new CodeVariableDeclarationStatement(typeof(Point),
"myPoint");
```

Als nächstes wird ein Objekt des Typs **Point** erzeugt, das benutzt wird um die Variable myPoint zu initialisieren.

Es werden hier die X- und Y-Koordinaten in einem CodePrimitiveExpression Objekt als primitive Datentypen dargestellt

```
CodeExpression parameters = new CodeExpression[]
{ new CodePrimitiveExpression(10), new CodePrimitiveExpression(10)  };
```

Es wird eine Instanz des Typs Point mit den Parametern x und y erstellt.

```
CodeObjectCreateExpression initializer;
initializer = new CodeObjectCreateExpression(typeof(Point), parameters);
```

Schließlich wird dem initializer, also die Instanz des Typs **Point,** die variablen Deklaration zugewiesen

```
myPointVariable.InitExpression = initializer;
```

So kann also jeder Programmiercode in Form einer CodeDom Struktur, die sprachunabhängig ist, beschrieben werden. Mit Hilfe von CodeDom kann Quellprogramm sprachunabhängig dargestellt werden.

Für die Serialisierung des Zustandsdiagramms in SUIT ist es jedoch nicht notwendig, eine CodeDom Struktur selbst zu erzeugen. .Net bietet hierfür in **System.ComponentModel. Design.Serialization-Namespace** Klassen an. **Component** implementiert Klassen von **System.ComponentModel.Design.Serialization-Namespace**. In SUIT sind beide Klassen **Zustandsdiagramm** und **ZustandsdiagrammElement** von **Component** abgeleitet und erben somit die Funktionalität der von **Component** implementierten Klassen.

Für die Überführung der **CodeDom** Struktur in Programmiercode dienen die in den **System.CodeDom.Compiler-Namespace** bereitgestellten Funktionen. **System.CodeDom.Com-piler-Namespace** enthält Klassen für das Generieren und Kompilieren von Quellcode in einer unterstützten Programmiersprache. Das Zustandsdiagramm ist in SUIT Sprachunabhängig realisiert. Dies wird durch die Benutzung

von **CodeDom** von .Net ermöglicht. Im Folgenden wird ein Ausschnitt des in SUIT generierten Codes für das Zustandsdiagramm aus Abb. 30 gezeigt:

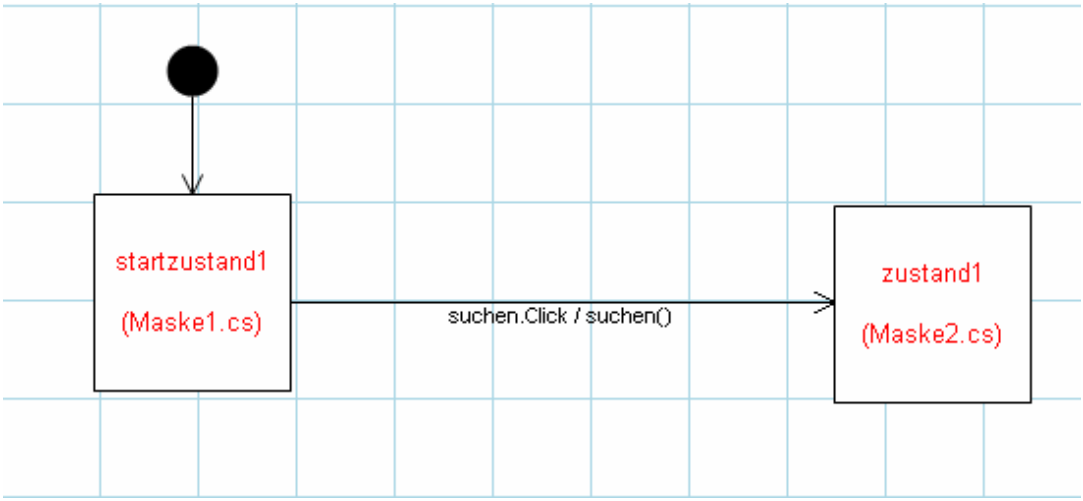


Abb. 30 Zustandsdiagramm

```
public class Beispiel : SUIT.ZustandsdiagrammEditor.Zustandsdiagramm
      {
  private SUIT.ZustandsdiagrammEditor.Startzustand startzustand1;
  private SUIT.ZustandsdiagrammEditor.Zustand zustand1;
  private SUIT.ZustandsdiagrammEditor.Zustandsuebergang zustandsuebergang1;

  #region Zustandsdiagramm Designer generated code

 private void InitializeComponent()
           {
 this.startzustand1 = new SUIT.ZustandsdiagrammEditor.Startzustand();
 this.zustand1 = new SUIT.ZustandsdiagrammEditor.Zustand();
 this.zustandsuebergang1 = new
      SUIT.ZustandsdiagrammEditor.Zustandsuebergang();



  // startzustand1
  this.startzustand1.BackColor = System.Drawing.Color.White;
  this.startzustand1.Bezeichnung = "startzustand1";
  this.startzustand1.BorderColor = System.Drawing.Color.Black;
  this.startzustand1.Groesse = new System.Drawing.Size(100, 100);
  this.startzustand1.GUIReferenz = "Fallstudie-CSharp:Maske1.cs";
  this.startzustand1.Info = null;
...
```

So sieht am Beispiel des Startzustands aus Abb. 30 der in SUIT automatisch generierte Code in C# aus.

CodeDom ist jedoch mit der jetzigen Version für die sprachunabhängigen Generierung der Fassade nicht ausreichend, da das CodeDom Model im vorliegenden .Net Framework Version 1.1 für Objekte, die nicht von Typ IComponent sind, noch nicht komplette Sprachunabhängigkeit realisiert. Für die Fassadengenerierung wird die Klasse **FileCodeModel** benutzt. Diese ermöglicht den Zugriff auf die Methoden innerhalb der Fassade. **FileCodeModel** ist im **EnvDTE-Namespace** definiert. Folgender Codeausschnitt zeigt die zu dem Beispiel aus Abb. 30 generierte Fassade.

```
namespace WindowsApplication3
{
    /// <summary>
    /// Zusammenfassung für Fassade.
    /// </summary>
    public class Fassade
        {
        public static void suchen(object sender, System.EventArgs e)
        {
            Output.WriteLine("suchen() invoked.");
         }
        // <summary>
        /// Code generated by FassadeGenerator. Don't change next
        /// statement.
        /// Change to this statement may cause exception if the code is
        ///regenerated.
        /// </summary>
        public static System.IO.TextWriter Output = System.Console.Out;

        public Fassade()
              {
                    //
                    // TODO: Fügen Sie hier die Konstruktorlogik hinzu
                    //
              }
        }
}
```

Wie im Code ersichtlich wird, wurde für das Zustandsdiagramm aus Abb. 30 eine Fassade automatisch generiert. Das Zustandsdiagramm besteht aus einem Zustandsübergang vom Startzustand1 zu Zustand1. Auf dem Zustandsübergang ist die Aktion suchen() definiert. Die Aktion findet man als Methodenrumpf in der Fassade wieder.

6.4.4 Das Anzeigen von Eigenschaften in einem Eigenschaftsfenster

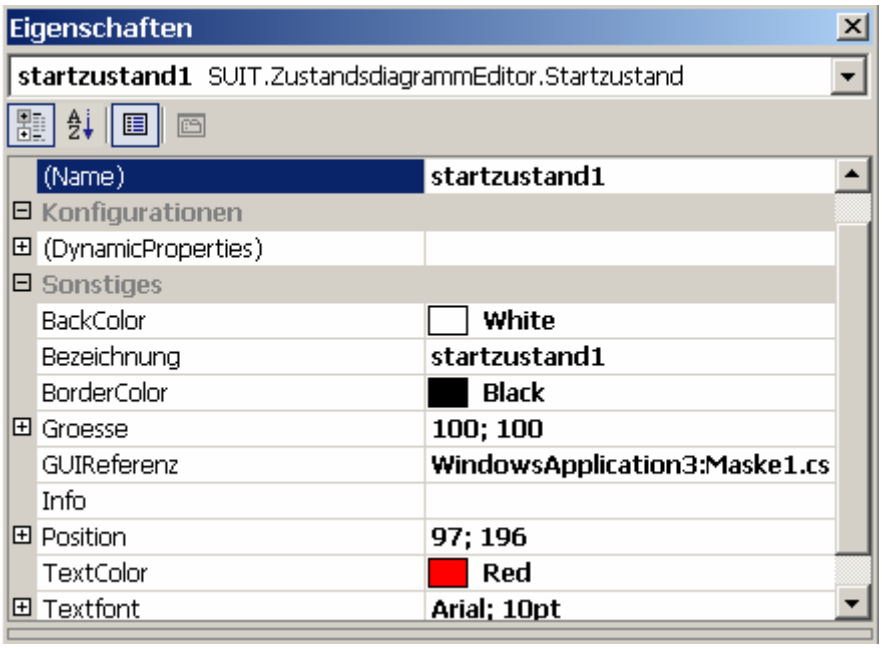


Abb. 31 Eigenschaftsfenster im Zustandsdiagrammeditor

Instance Variablen, die über getter- und setter-Methoden verfügen, werden standardmäßig als Eigenschaften im Eigenschaftsfenster angezeigt. Mit der **BrowsableAttribute**-Klasse kann definiert werden, ob eine Eigenschaft im Eigenschaftsfenster (Abb. 31) angezeigt werden soll. Der Zustandsdiagrammeditor zeigt im Eigenschaftsfenster die Eigenschaften nicht an, die mit dem **Browsable**-Konstruktor mit dem Wert **false** gekennzeichnet sind. Standardmäßig ist jede Eigenschaft mit dem **BrowsableAttribute**-Konstruktor mit dem Wert **true** markiert. Werden Eigenschaften im Eigenschaftsfenster (Abb. 31) angezeigt, so kann für jede Eigenschaft definiert werden, ob es sich bei dieser Eigenschaft um eine schreibgeschützte Eigenschaft oder um eine Lese-/Schreib-Eigenschaft handelt. Eigenschaften, die den **System.Component-Model.ReadOnlyAttribute** -Konstruktor mit dem Wert **true** versehen sind, sind schreibgeschützt.

6.4.5 Aufrufen des GUI-Designers in SUIT

EnvDTE DTE

DTE dient zur Automatisierung und Erweiterbarkeit von Visual Studio. Mit einem DTE Object können Prozesse in Visual Studio, wie z.B. das Kompilieren von Projekten und der Zugriff auf Elemente der Entwicklungsumgebung wie z.B. der Zugriff auf den GUI-Builder von Visual Studio automatisiert werden.

DTE am Beispiel des Aufruf des GUI-Builder in SUIT

Der Aufruf des GUI-Builders in SUIT ist notwendig, da z.B. bei einem Doppelklick auf einem Zustand der GUI-Builder mit der dem Zustand zugeordneten Maske geöffnet wird. Diese kann dann bearbeitet werden.

Für den Aufruf des GUI-Builders benötigt man ein Objekt der Enwticklungsumgebung, angeboten durch das **DTE**-Objekt im Namespace **EnvDTE**. Ein Objekt des Typs **DTE** repräsentiert unter anderem alle bestehenden Projekte mit den darin enthaltenen Dateien, alle Konstanten etc. innerhalb der Entwicklungsumgebung. Ist das **DTE**-Objekt erst einmal bei der Hand, kann man sich das Projekt holen, das die GUI-Datei enthält. Hat man nun das Projekt in der sich die GUI-Datei befindet, so kann man sich ein Objekt der **EnvDTE.ProjectItem** erzeugen, der auf die GUI-Datei zeigt.

ProjectItem-Schnittstelle stellt ein Element in einem Projekt dar, in unserem Falle ist es die GUI-Datei, die beim Doppelklick angezeigt werden soll. Hat man nun das Projektitem, also das Element aus dem Projekt, hier die GUI-Datei, so kann diese geöffnet werden. Soll nun diese GUI-Datei in der Ansicht des GUI-Builders geöffnet werden, kann dies mit Hilfe der Klasse **Constants** im Namespace **EnvDTE** definiert werden.

Die Klasse **Constants** definiert *vsViewKindDesigner*, um eine Datei in der Designer Ansicht, also in der Ansicht des GUI-Builders anzuzeigen. Folgender PseudoCode zeigt diese Vorgehensweise.

Zuerst holt man sich das aktuelle Projekt. Als Parameter werden der Pseudo_Methode getProject den Namen der GUI-Datei und einen Objekt der Klasse EnvDTE.DTE myDTE

übergeben. myDTE enthält alle bestehenden Projekte der Entwicklungsumgebung Visual Studio.Net.

```
EnvDTE.Project pr = getProject(GUI_Datei_Name, myDTE);
```

Hat man nun das aktuelle Projekt, so erzeugt man ein Objekt der Klasse ProjectItem, das die zu öffnende GUI-Datei als Element darstellt.

```
EnvDTE.ProjectItem myGUI = getProjektItem(GUI-Datei-Name);
```

Hat man nun ein Projektitem der GUI erzeugt, so kann dieses in der GUI-Builder Ansicht geöffnet werden.

```
myGUI.Open(EnvDTE.Constants.vsViewKindDesigner);
```

6.5 Schnittstelle von SUIT

Um SUIT um ein weiteres Zustandsdiagrammelement zu erweitern, muss

- eine von **ZustandsdiagrammElement** abgeleitete Komponente
- eine von **ZDiagrammElementDesigner** abgeleitete Designerklasse

implementiert werden.

6.5.1 ZustandsdiagrammElement

ZustandsdiagrammElement ist eine abstrakte Klasse. Sie stellt eine Komponente dar. Um SUIT um neue Komponenten erweitern zu können, muss diese Klasse geerbt werden. Die Schnittstelle ZustandsdiagrammElement implementiert folgenden abstrakten Methoden:

draw

```
public abstract void draw(Graphics g);
```

Diese Methode *draw* dient zum Zeichnen von der Komponente.

Parameter	Beschreibung
Graphics g	Das Graphics Objekt, auf die die Komponente zu zeichnen ist

Tabelle 5 Parameter von draw

DrawCreateDragFeedback

```
public abstract void DrawCreateDragFeedback(ZustandsdiagrammElement start-
Element, Point startPoint, ZustandsdiagrammElement endElement, Point
endPoint, Point myScreenOffset, Color backColor, Point scrollPos, float
zoom);
```

Die Methode *DrawCreateDragFeedback* ermöglicht dem Benutzer während einer Drag & Drop Operation beim Erzeugen einer Komponente ein Visuelles Feedback anzuzeigen. Das Rechteck auf Abb. 32 visualisiert ein Feedback beim Erzeugen eines Zustands per Drag&Drop.

DrawCreateDragFeedback sollte so zeichnen, das der erste Aufruf das Feedback an der angegebenen Position zeichnet, und der zweite Aufruf es wieder löscht.

In *DrawCreateDragFeedback* bietet es sich an, die Methoden von **ControlPaint** zu benutzen. **ControlPaint** bietet Methoden wie *DrawReversibleFrame* und *DrawReversibleLine*. DrawReversibleFrame zeichnet einen umkehrbaren Rahmen auf dem Bildschirm, während *DrawReversibleLine* eine umkehrbare Linie auf dem Bildschirm zeichnet.

Die *DrawRevisible* Methoden von ControlPaint nutzen beim Zeichnen die globalen Koordinaten einer Komponente. Die globalen Koordinaten beschreiben die Lage der graphischen Komponenten aus der Sicht des Anwenders. Die globalen Koordinaten sind bezogen auf die gesamte Bildschirmfläche. In *DrawCreateDragFeedback* sollte deshalb zusätzlich noch das Offset des Bildschirms, der aktuelle Zoomfaktor und die Scrollposition berücksichtigt werden, um aus den relativen Koordinaten, diese sind nur auf die Zeichenfläche bezogen, die globalen Koordinatenwerte umzurechnen. Siehe dazu Abb. 32.

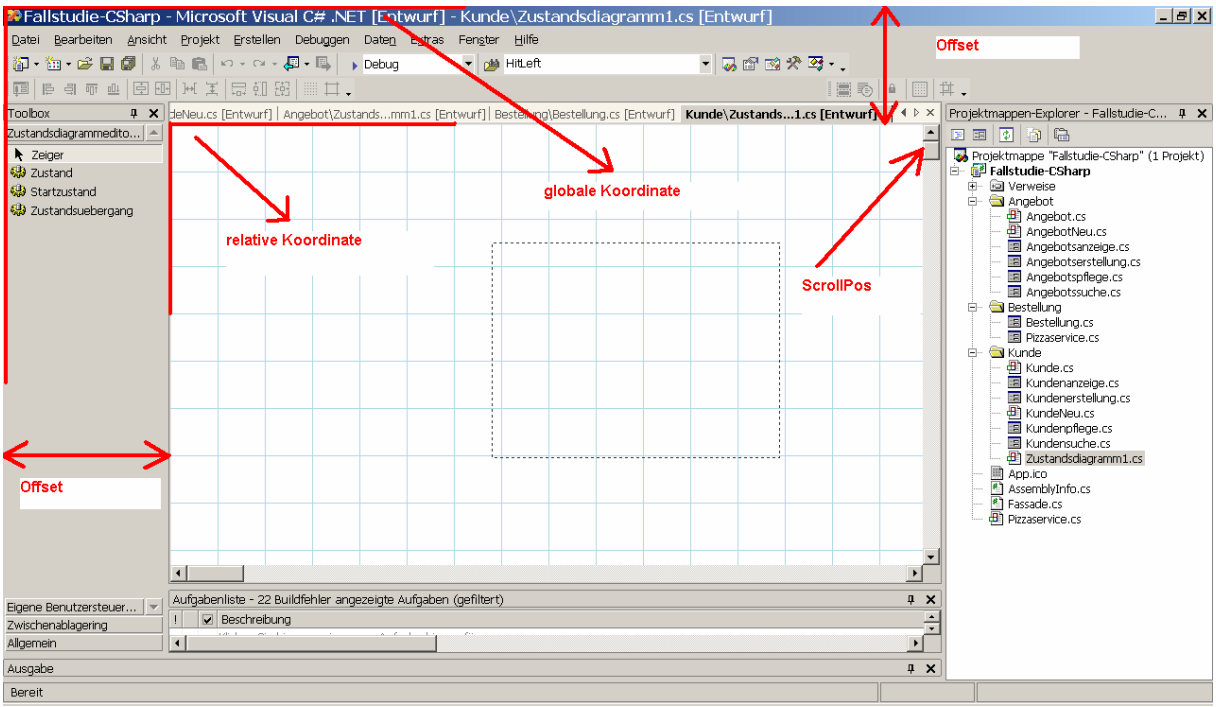


Abb. 32 Visual Studio.net in der Ansicht von SUIT

Parameter	Beschreibung
ZustandsdiagrammElement startElement	Das ZustandsdiagrammElement, das an der Position des startPoint angesiedet ist. Ist null, wenn an den startPoint kein Element vorhanden ist.
Point startPoint	Der Start Punkt, an dem der Drag begann. Dieser Punkt ist dem DragControl gebunden und nicht global, daher muss er noch mit screenOffset addiert werden.
ZustandsdiagrammElement endElement	Das ZustandsdiagrammElement, das an der Position des endPoint angesiedet ist. Ist null, wenn an den endPoint kein Element vorhanden ist
Point endPoint	Der End-Punkt, an dem der Drag endet. Dieser Punkt ist dem DragControl gebunden und nicht global, daher muss er noch mit screenOffset addiert werden.
Point myScreenOffset	Dieses Offset kann zu den Point-Koordinaten addiert werden, um sie zu den globalen Koordinaten zu konvertieren, so dass mit diesen Koordinaten die *DrawReversible* Methode in ControlPaint direkt verwendet werden kann.
Point scrollPos	Die Scroll-Position, auf die umgerechnet werden soll. Diese Position ist bereits vom Zoom Faktor berücksichtig worden und kann einfach zum Punkt addiert werden.
Float Zoom	der Zoom Faktor, auf der umgerechnet werden soll.
Color backColor	Die Hintergrundfarbe, die hinter den Elementen sind. Wird oft gebraucht für das *Reverisble Feedback.*

Tabelle 6 Parameter von DrawCreateDragFeedback

delete

```
public virtual void delete(Zustandsdiagramm myDiagram, IDesignerHost
myHost)
```

Die Methode *delete* ist *virtual*, d.h. sie kann in einer abgeleiteten Klasse überschrieben werden. *delete* dient dem Löschen einer Komponente. Das Löschen ist Komponentenabhängig, so müssen z.B. beim Löschen eines Zustands alle hinein- und hinausführenden Zustandsübergänge mitgelöscht werden.

Wird die Methode delete in der Unterklasse nicht überschrieben, so wird die Funktionalität von delete geerbt. Die vererbte delete-Methode bewirkt, dass eine Komponente nur sich selbst löscht.

Parameter	Beschreibung
Zustandsdiagramm myDiagram	Das Zustandsdiagramm Object von dem die Komponente gelöscht werden soll.
IDesignerHost myHost	myHost ist ein Objekt zum Verwalten von Designertransaktionen und Komponenten und Komponenten

Tabelle 7 Parameter von delete

initialize

public abstract bool initialize(Design.ZustandsdiagrammDesigner designer, String name, Point startPunkt, Point endPunkt, ZustandsdiagrammElement startElement, ZustandsdiagrammElement endElement);

initialize dient der Initialisierung einer Komponente bei ihrer Erzeugung

Parameter	Beschreibung
ZustandsdiagrammDesigner designer	Ein Designer des Typs ZustandsdiagrammDesigner
String name	Name der Komponente
Point startPunkt	Der Start Punkt, an dem der Drag begann. Dieser Punkt ist dem DragControl gebunden und nicht global, daher muss er noch mit screenOffset addiert werden.
Point endPunkt	Der End-Punkt, an dem der Drag endet. Dieser Punkt ist dem DragControl gebunden und nicht global, daher muss er noch mit screenOffset addiert werden.
ZustandsdiagrammElement startElement	Das ZustandsgiagrammElement, das an der Position des startPoint angesiedet ist. Ist null, wenn an dem startPoint kein Element vorhanden ist.
ZustandsdiagrammElement endElement	Das ZustandsgiagrammElement, das an der Position des endPoint angesiedelt ist. Ist null, wenn an dem endPoint kein Element vorhanden ist.

Tabelle 8 Parameter von initialize

6.5.2 ZDiagrammElementDesigner

Die abstrakte Klasse **ZDiagrammElementDesigner** stellt eine Basisdesignerklasse zum Erweitern des Verhaltens einer Komponente im Entwurfsmodus dar. **ZDiagrammElementDesigner** implementiert folgende Static-Variablen:

Static- Variable	Beschreibung
Object HitMove	Wird gesetzt, wenn die Maus einen Punkt – meist irgendwo in der Mitte – des ZustandsdiagrammElement berührt hat.
Object HitBottom	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am unteren Rand berührt hat.
object HitLeft	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am linken Rand berührt hat.
object HitRight	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am rechten Rand berührt hat.
object HitTop	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am oberen Rand berührt hat.
object HitLowerLeft	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am unteren linken Rand berührt hat.
object HitLowerRight	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am unteren rechten Rand berührt hat.
object HitUpperLeft	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am oberen linken Rand berührt hat.
object HitUpperRight	Wird gesetzt, wenn die Maus ein ZustandsdiagrammElement am oberen rechten Rand berührt hat.
Size myAdornmentDimensions	Liefert die Größe des Verzierungvierecks, das beim Selektieren eines Elementes in Designer gezeichnet werden soll.

Tabelle 9 Statische Variablen der Klasse ZDiagrammElementDesigner

Wird eine Klasse von **ZDiagrammElementDesigner** abgeleitet muss die folgende Methode implementieren:

BoundingBox

```
public abstract Rectangle BoundingBox { get; set;}
```

Dieser Property gibt einen Rechteckigen Bereich für das Element zurück, anhand dessen ermittelt wird, welcher Bereich neu gezeichnet werden muss, wenn sich am Element bzw. in seiner Darstellung etwas verändert. Dies kann z.B. beim Selektieren durch zeichnen eines Selektionsrahmen erfolgen.

GetHitTest

```
public abstract object GetHitTest(int x, int y);
```

Diese Methode überprüft, ob das Element mit dem Mauszeiger berührt wurde. Der Root-Designer wird jedes Mal diese Methode aufrufen, wenn ein Maus-Move-Ereignis ausgelöst wird. Das zurückgegebene Objekt ist Null, wenn die Koordinaten nicht auf dem Element liegen. Wenn die Koordinaten auf dem Element sind, gibt die Methode ein Identifizierungsobjekt wieder, dass der Designer später weiter benutzen kann. Dieses Rückgabeobjekt kann der *GetCursor*-, *DrawDragFeedback*- und *Drag*-Methode übergeben werden.
Die Methode *GetHitTest* gibt ein Object (Tabelle 4) zurück, dass die Berührung identifiziert. Dieser kann null sein, wenn der Punkt sich nicht im Element befindet.

Parameter	Beschreibung
Int x	Die X Koordinate für den Berührungstest.
Int y	Die Y Koordinate für den Berührungstest.

Tabelle 10 Parameter von GetHitTest

IsInRect

```
public abstract bool IsInRect(Rectangle r);
```

Diese Methode überprüft, ob eine Komponente in einem übergebenden Rechteck r enthalten ist. Diese Methode wird bei einer Selektion durch einen rechteckigen Bereich, das per Drag&Drop gezogen worden ist, verwendet, um zu überprüfen, ob das Element in diesem Rechteck enthalten ist.

Sie gibt einen Boolean Wert zurück, der *true* ist, wenn das Element in dem übergebenden Rechteck r enthalten ist, ansonsten *false*.

Parameter	Beschreibung
Rectangle r	Das Rechteck, das den Bereich beschreibt, in dem überprüft wird, ob dieses Element in diesem Bereich enthalten ist.

Tabelle 11 Parameter von IsInRect

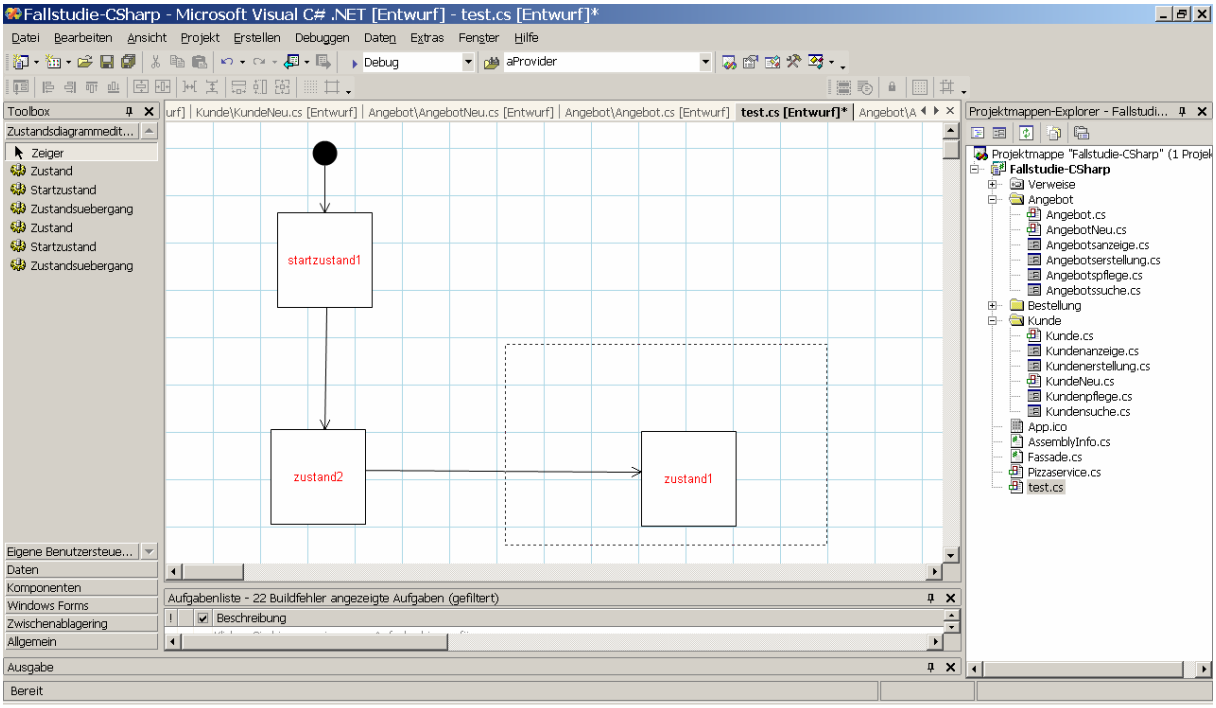


Abb. 33 SUIT-Zustandsdiagramm mit DragFeedback

Die Komponente Zustand mit dem Namen zustand1 würde bei einem Methodenaufruf von *IsInRect* true liefern, weil sie sich im Rechteck befindet. Dagegen liefert das Komponenten Object mit dem Namen zustand2 bei *IsInRect* false, weil sich zustand2 außerhalb des rechteckigen Bereichs befindet.

GetCursor

```
public virtual Cursor GetCursor(object hitObject)
```

GetCursor gibt den Maus-Cursor für den übergebenden HitObjekt zurück. Handelt es sich bei dem übergebenen hitObject um HitLeft oder HitRight so wird ein Cursor Object zurückgegeben mit zwei horizontalen Spitzen (West/Ost). Ist es aber ein *HitTop* oder *HitBottom* Object, gibt *GetCursor* ein Cursor Object mit zwei vertikalen Spitzen (Nord/Süd). Bei einem *HitUpperLeft* oder *HitLowerRight* ist es ein Cursor mit zwei diagonalen Spitzen (Nordwest/Südost).Handelt es sich um *HitUpperRight* oder *HitLowerLeft* ist der Cursor zwei diagonale Spitzen(Nordost/Südwest). Ansonsten gibt die Methode GetCursor einen Cursor Object mit vier Spitzen, die nach Nord, Süd, Ost und West zeigen, zurück (siehe Tabelle 8).

Die Methode *GetCursor* ist *virtual* und kann in einer von **ZDiagrammElementDesigner** abgeleiteten Klasse überschrieben werden.

Parameter	**Beschreibung**
object hitObject	Ein Objekt, dass von einer GetHitTest Methode zurückgeliefert wurde.

Tabelle 12 Parameter von GetCursor

DrawAdornments

```
public abstract void DrawAdornments(Graphics g, bool primary);
```

Die Methode *DrawAdornments* wird aufgerufen, wenn das Element selektiert wurde. Diese Methode zeichnet den Selektionsrahmen um das Element.

Parameter	**Beschreibung**
Graphics g	Das Graphics Objekt, auf das der Selektionsrahmen zu zeichnen ist.
Bool primary	primary ist true, wenn das selektierte Element das primäre Selektierte Element ist. Dieser Parameter ist bei Mehrfachauswahl Interessant.

Tabelle 13 Parameter von DrawAdornments

DrawDragFeedback

```
public abstract void DrawDragFeedback (object hitObject, Color backColor,
Point screenOffset, Point mouseOffset, Point scrollPos, float zoom, Point
initialPoint);
```

DrawDragFeedback wird benutzt, um den Drag Feedback. (Abb. 34) beim Resizing für den User aufzuzeichnen. *DrawFeedback* sollte so zeichnen, das der erste Aufruf das Feedback an der angegebenen Position zeichnet, und der zweite Aufruf es wieder löscht. Siehe dazu *DrawCreateDragFeedback.* Hier können die Methoden der Klasse **ControlPaint** benutzt werden, da diese Methoden wie *DrawReversibleLine* oder *DraReversibleFrame* unterstützt.

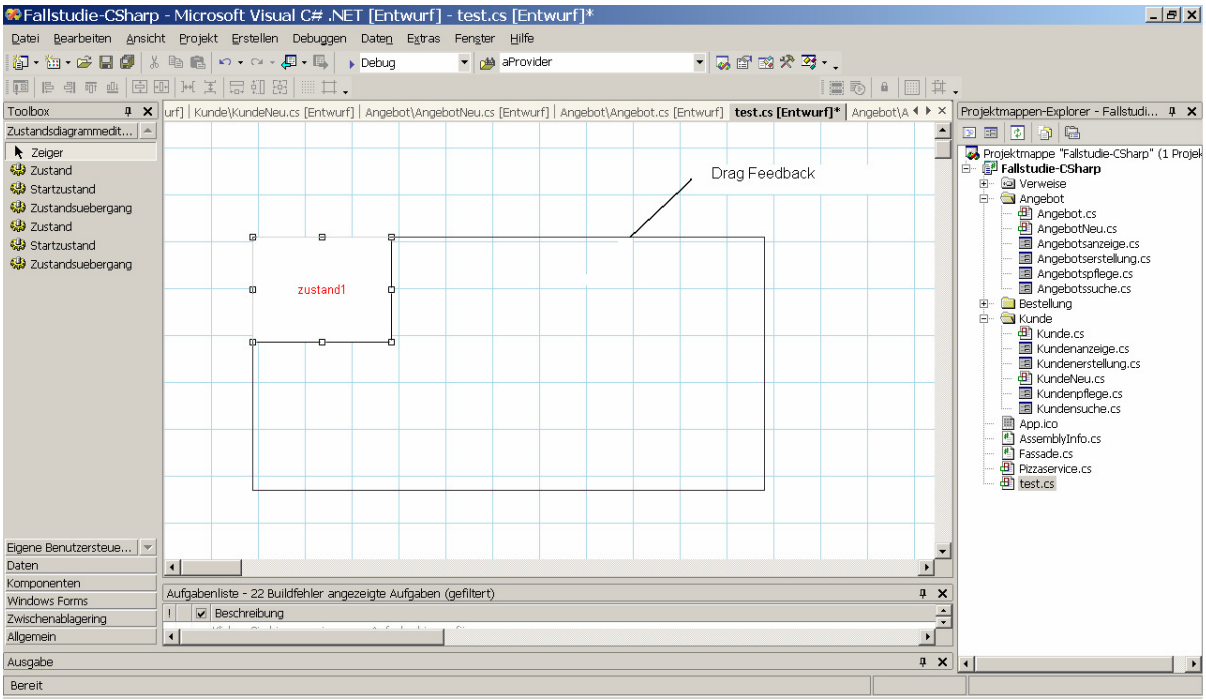


Abb. 34 Dragfeedback

Parameter	Beschreibung
object hitObject	Das zurückgebene Objekt einer zuvor aufgerufenen *GetHitTest*-Methode.
Color backColor	Die Hintergrundfarbe, die hinter den Elementen ist. Wird oft gebraucht für das Reversible Feedback.
Point screenOffset	Dieses Offset kann zu den Elementkoordinaten addiert werden, um sie zu den globalen Koordinaten zu konvertieren, so dass mit diesen Koordinaten die *DrawReversible* Methode in **ControlPaint** direkt verwendet werden kann.
Point mouseOffset	Das aktuelle Offset von der Maus zur der Position, wo der Drag begonnen hat.
Point scrollPos	Die Scroll-Position, auf die umgerechnet werden soll. Diese Position ist bereits vom Zoomfaktor berücksichtigt worden und kann einfach zum Punkt addiert werden.
Float Zoom	der Zoomfaktor, auf den umgerechnet werden soll.
Point initialPoint	Der Punkt, an dem dieser Drag initialsiert wurde zu dem Zustandselement in der Y Richtung

Tabelle 14 Parameter von DrawDragFeedback

Drag

```
public abstract void Drag(object hitObject, int x, int y, Point
initialPoint);
```

Drag wird aufgerufen am Ende eines erfolgreichen Drags um die neue Position/Größe eines Zustandselements zu aktualisieren. Die X und Y Werte sind relativ zu dem Drag, so dass sie einfach zu den Koordinaten des Komponents hinzuaddiert werden können. Das Objekt, dass von *GetHitTest* zurückgeliefert wurde, wird weitergegeben, so dass die Methode *Drag* weiß, ob die Komponente vergrößert, verschoben usw. werden soll.

Parameter	Beschreibung
object hitObject	Das Objekt, das von einem zuvor aufgerufenen *GetHitTest* zurückgeben wurde.
Int X	Die Offset-Anzahl zu dem Zustandselement in der X Richtung.
Int Y	Die Offset-Anzahl zu den Zustandselement in der Y Richtung
Point initialPoint	Der Punkt, an dem dieser Drag initialsiert wurde zu den Zustandselement in der Y Richtung

Tabelle 15 Parameter von Drag

onClick

```
public virtual void onClick(MouseEventArgs me, EnvDTE.DTE DTE)
```

onClick wird ausgeführt, wenn auf das Zustandsdiagrammelement geklickt wurde. onClick ist virtual und kann in einer von **ZDiagrammElementDesigner** abgeleiteten Klasse überschrieben werden.

Parameter	**Beschreibung**
MouseEventArgs me	Die Argumente für den Mouse-Event.
EnvDTE.DTE DTE	Der DTE des hostest.

Tabelle 16 Parameter von onClick

6.6 Fazit

Die in Kapitel 4 beschriebenen funktionalen Anforderungen an SUIT wurden bewertet, ob sie als erfüllt oder nicht erfüllt gelten können. Weiterhin wurden die technischen Besonderheiten bei der Umsetzung von SUIT in .Net beschrieben; damit wurde ein kleiner Einblick in .Net gewährt. SUIT kann weiterentwickelt werden. Zu diesem Zweck wurden die Schnittstellen zu SUIT beschrieben.

7. Fallstudie

7.1. Einleitung

Ziel der vorliegenden Bachelor-Arbeit ist die Entwicklung eines Tools, mit dem auf der Basis von Zustandsdiagrammen und Fensterentwürfen eine Benutzungsschnittstelle entwickelt und prototypisch implementiert werden kann.

Um die Eignung des Tools für die prototypische Implementierung einer Benutzungsschnittstelle zu demonstrieren, wird in diesem Kapitel eine Fallstudie durchgeführt. Die Fallstudie hat die Entwicklung der Benutzungsschnittstelle eines Pizzaservicesystems zum Thema.

Die Untersuchung hat zwei Ziele:

- Es soll demonstriert werden, wie das Tool für das Entwerfen von Benutzungsschnittstellen mit Hilfe von Zustandsdiagrammen und Fensterentwürfen eingesetzt werden kann. Es ist aufzuzeigen, wie der Prototyp einer Benutzungsschnittstelle schrittweise, lückenlos und ohne den Einsatz zusätzlicher Tools erzeugt werden kann.

- Durch die Demonstration anhand einer Fallstudie soll die Praxistauglichkeit des zu entwickelnden Tools überprüft werden.

Die Untersuchung unterliegt zugleich einer Beschränkung des Anspruchs. Es ist nicht das Ziel, ein lauffähiges System zu entwickeln, d.h. ein ausgereiftes Pizzaservicesystem zu entwerfen, das auch tatsächlich eingesetzt werden kann. Daher wird auf eine detaillierte Anforderungsanalyse verzichtet, die in der Praxis bei einem Software-Entwicklungsprozess unabdingbar wäre. Eine ausführliche Analyse der Fallstudie ist also im Rahmen dieses Kapitels nicht geplant. Eine detaillierte Beschreibung der Funktionalität des zu implementierenden Systems Pizzaservice besitzt keine direkte Relevanz für die in diesem Kapitel zu beschreibende Vorgehensweise.

Die Untersuchung wird in folgenden Schritten vorgenommen:

Im Abschnitt 7.2 wird eine allgemeine Beschreibung des Ist-Zustandes des Pizzaservicesystems gegeben. Daraufhin folgt im Abschnitt 7.3, ausgehend von einem Anwendungsfalldiagrammes des Ist- Zustandes, die schrittweise Erzeugung des Prototyps. Mit einem Fazit im Abschnitt 7.4 wird das Kapitel Fallstudie abgeschlossen.

7.2 Ist-Analyse

Das System in dieser Fallstudie verwaltet den Bestellvorgang eines Pizzaservices. Ein Kunde, der eine Pizza bestellen will, ruft bei dem Pizzaservice an, um seine Bestellung abzugeben.

Der Mitarbeiter, der diesen Anruf entgegen nimmt, soll mit Hilfe des Fallstudie-Systems alle Bestellungen überblicken und die Bestellung des Kunden aufnehmen können.

Ruft ein Kunde an, wird, bevor die Bestellung entgegen genommen wird, überprüft, ob er dem System bekannt ist. Dies geschieht durch die Anfrage der Haustelefonnummer des Kunden. Die Haustelefonnummer wird als Identifikationskriterium gewählt, weil sie die Anschrift, und somit die Lieferadresse, eher eindeutig identifiziert als es über den Namen der Fall ist, da in einem Haushalt, und somit bei einer Lieferadresse, mehrere Personen wohnen können. Der „Kunde“ ist so gesehen weniger die Person, die die Bestellung aufgegeben hat, als der Ort, an den die Bestellung ausgeliefert wird. Um ein schnelles Auffinden von im System bereits gespeicherten Kunden-Adressen zu gewährleisten, ohne dabei immer wieder erneut die Adresse einzugeben, wird daher auf die Haustelefonnummer zurückgegriffen. Außerdem dient sie für Rückrufe an den Kunden. Eine Kundennummer ist bei einem Pizzaservice-System nicht praxistauglich und wird daher nicht verwendet.

Ist der Kunde nicht im System bekannt, werden seine Kundendaten, wie z.B. Anschrift, Name etc., angefragt und dem System vermittelt. Ist der Kunde schon im System registriert, können gegebenenfalls die Kundendaten gepflegt und bei Änderungen aktualisiert werden.

Nun kann der Mitarbeiter die Bestellungen des Kunden entgegen nehmen. Dabei überprüft das System, ob die gewünschten Artikel vorhanden sind.

Ist die Bestellung abgegeben, wird diese in Form einer Rechnung ausgedruckt, die an die Küche weitergereicht wird. Hier werden die bestellten Pizzen gemäß der weitergereichten Rechnung gebacken, auslieferungsreif verpackt und samt Rechnung einem Fahrer zum Ausliefern weitergereicht.

Konnte eine Auslieferung aus irgendeinem Grund nicht erfolgreich abgewickelt werden, z.B. weil der Kunde nicht angetroffen wurde, wird dem System mitgeteilt, dass die Auslieferung nicht erfolgreich abgeschlossen werden konnte.

Darüber hinaus muss das System dem Anwender eine Möglichkeit bieten, seine Stammdaten zu pflegen.

7.3 Schrittweise Erzeugung des Prototyps

7.3.1 Vorgehensweise

Die Entwicklung des Prototyps erfolgt unter Einbeziehung von Konzepten des WAM-Ansatzes [Züllighoven 1998]. Es wird zunächst unter Verwendung der Metapher Werkzeug und Material des WAM-Ansatzes versucht, die Materialien zu erkennen und das Gesamtsystem in einzelne Werkzeuge aufzuteilen. Die Werkzeug und Material Metapher des WAM-Ansatzes hat sich bei den immer komplexer werdenden Anwendungen als sehr fruchtbar erwiesen.

Denn je größer ein Softwaresystem ist, desto schwieriger wird es für den einzelnen, das Ganze komplett zu überblicken. Somit würde die Untergliederung eines Gesamtsystems in Werkzeugen die Verständlichkeit erhöhen und die Komplexität reduzieren.

Der detaillierte Entwurf von Materialien ist in diesem Abschnitt kein primäres Ziel. Der Schwerpunkt liegt auf der interaktiven Präsentation und Handhabung des Materials. Diese fallen in die Zuständigkeit des Werkzeugs. Ein Werkzeug präsentiert Materialien gemäß ihrem Verwendungszusammenhang und ermöglicht ihre Sondierung und Manipulation. Es vergegenständlicht immer wiederkehrende Arbeitstätigkeiten, welche an Materialien ausgeführt werden. Materialien können nach [Züllighoven 1998] sondiert und manipuliert werden. Sie werden durch Werkzeuge bearbeitet.

Da es sich hier nicht um ein komplexes System handelt, bietet das Anwendungsfalldiagramm ausreichend Anhaltspunkte für Werkzeuge, so dass in diesem Abschnitt auf die Beschreibung von Szenarien verzichtet wird.

Das Gesamtsystem (hier: Pizzaservicesystem) wird zunächst in einem Anwendungsfalldiagramm (siehe Abb. 35) dargestellt, in dem die Zusammenhänge zwischen den Anwendungsfällen untereinander und zwischen ihnen und den beteiligten Akteuren beschrieben werden. Dieses Diagramm bietet den Ausgangspunkt für den nächsten Schritt: der Identifizierung von Werkzeugen.

Um den Entwurf bzw. die Identifikation von Werkzeugen zu erleichtern, wird in [Züllighoven 1998] zusätzlich die Beschreibung von Szenarien empfohlen. Dies ist bei der vorliegenden Fallstudie nicht erforderlich, da es sich nicht um ein komplexes, schwer überschaubares System handelt. In unserem Beispiel bietet ein Anwendungsfalldiagramm ausreichend Anhaltspunkte, um Werkzeuge zu erkennen.

Im Folgenden werden einige weitere Begriffe gemäß [Züllighoven 1998, Seite 48] verwendet:

- „Ein einfaches Werkzeug besitzt anwendungsfachlich eine elementare Funktionalität. Softwaretechnisch besitzt es keine Sub-Werkzeuge.“
- „Ein Kombi-Werkzeug kombiniert verschiedene fachliche Dienstleistungen zur Erledigung einer komplexen Aufgabe. Softwaretechnisch ist es aus existierenden Sub-Werkzeugen zusammengesetzt.“
- „Ist ein Werkzeug als technische Komponente innerhalb eines anderen Werkzeugs eingesetzt, so wird es als Sub-Werkzeug bezeichnet.“

Nach ihrer Identifikation werden die Werkzeuge wiederum zu einem Gesamtsystem kombiniert.

Diese Vorgehensweise demonstriert, wie mit dem Tool der vorliegenden Arbeit eine Benutzungsschnittstelle für ein vollständiges System (Pizzaservicesystem) modelliert werden kann.

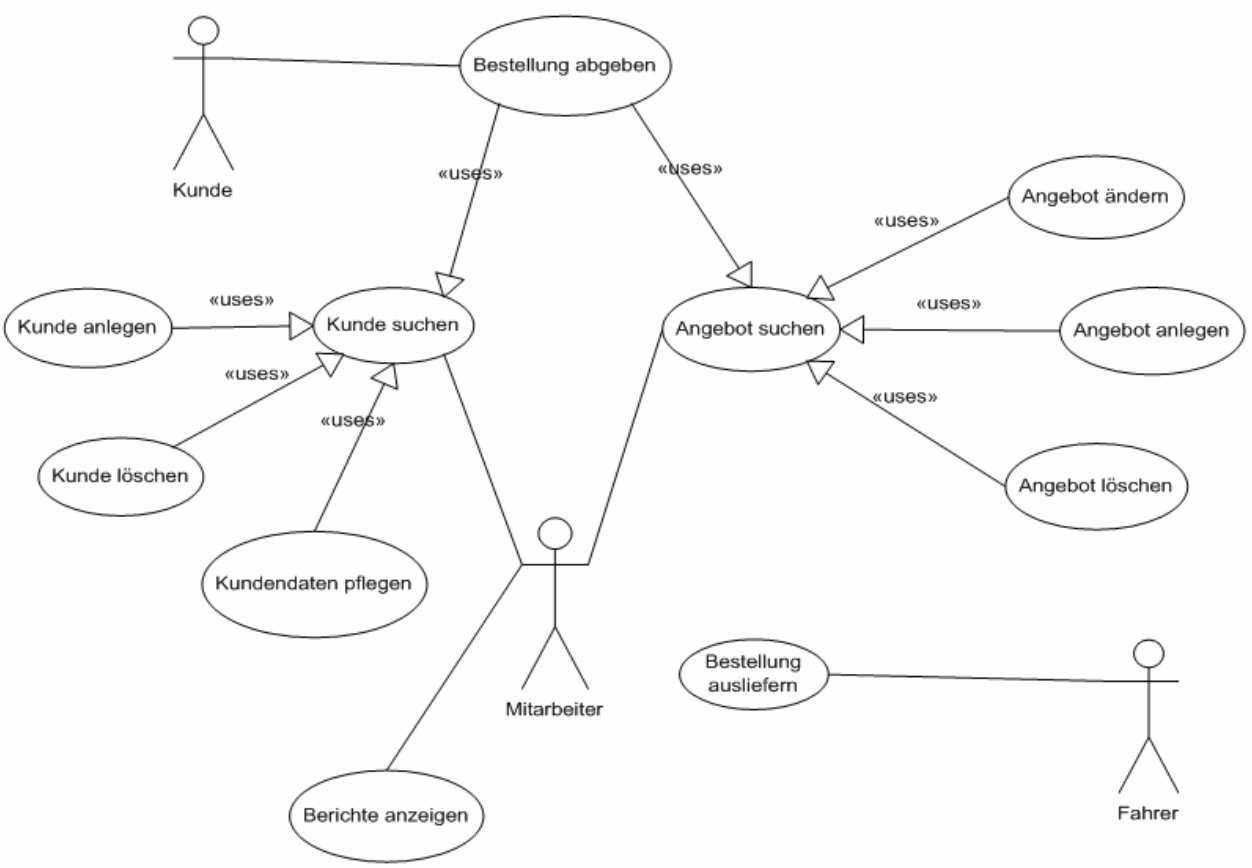


Abb. 35 Anwendungsfalldiagramm Pizzaservice

7.3.2. Identifikation der elementaren Werkzeuge

Das auf der vorgenommenen Ist-Analyse beruhende Anwendungsfalldiagramm des zu entwickelnden Pizzaservicesystems ist in Abb. 35 aufgeführt. Dem Anwendungsfalldiagramm kann eine Gruppierung, also gleichartige Handlungen an ähnlichen Datensätzen entnommen werden. Diese bilden die ersten Anhaltpunkte für Werkzeuge.

- Die Anwendungsfälle „Kunden suchen“, „Kunden anlegen“, „Kunden löschen“ und „Kunden pflegen“ sind zusammengehörige Anwendungsfälle; hier werden Kundendaten bearbeitet.
- Die Anwendungsfälle „Angebot suchen“, „Angebot ändern“, „Angebot anlegen“ und „Angebot löschen“ bearbeiten Angebotsdaten.

Dies legt nahe, Kundendaten und Angebotsdaten als Materialien zu betrachten. Dieser Betrachtungsweise nach können zwei elementare Werkzeuge Kundenverwaltungswerkzeug und Angebotsverwaltungswerkzeug identifiziert werden. Präsentation und Manipulation von Kundendaten und Angebotsdaten sind ausschließlich über diese Werkzeuge möglich. Kundendaten z.B. können über das Kundenverwaltungswerkzeug angezeigt, gepflegt, gelöscht und neu angelegt werden.

Bei den beiden identifizierten Werkzeugen handelt es sich um „einfache“ Werkzeuge mit elementarer Funktionalität, da sie wiederkehrende Arbeitsabläufe wie „Kunden suchen“ oder „Kundendaten pflegen“ unterstützen.

Nachdem die elementaren Werkzeuge Kundenverwaltungswerkzeug und Angebotsverwaltungswerkzeug identifiziert worden sind, widmen wir uns im Folgenden der graphischen Repräsentation der einzelnen Werkzeuge.

7.3.3 Entwicklung der Benutzungsschnittstelle auf Basis eines Zustandsdiagramms

Eine Benutzungsschnittstelle besteht in der Regel aus einer Menge von Masken und Dialogboxen, zwischen denen durch Eingabeaktionen (Menüauswahl, »OK«-Button usw.) hin und her gewechselt werden kann. In der zustandsbasierten Modellierung befindet sich die Benutzungsschnittstelle immer dann in einem Zustand, wenn sie auf eine Eingabe wartet. Tritt ein Ereignis z.B. durch das Betätigen einer Taste auf, so navigiert die Benutzungsschnittstelle von einem Zustand zu einem anderen Zustand. Werden also Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen modelliert, so kann jedem Zustand eine Maske zugeordnet werden. (siehe 1.2)

Im Folgenden wird dies zunächst für die identifizierten Werkzeuge „Angebotsverwaltungswerkzeug“ und „Kundenverwaltungswerkzeug“ in SUIT durchgeführt.

7.3.4 Angebotsverwaltungswerkzeug

Wie man dem Anwendungsfalldiagramm in Abb. 35 entnehmen kann, benutzen die Anwendungsfälle „Angebot ändern“, „Angebot anlegen“ und „Angebot löschen“ den Anwendungsfall „Angebot suchen“. Daher muss eine Angebotssuche ausgeführt werden, ehe eine weitere Aktion auf ein Angebot erfolgen kann. Damit ist eine bestimmte Reihenfolge vorgegeben.

Wird nun dieser Ablauf durch ein Zustandsdiagramm beschrieben, ergeben sich daraus vier Zustände, die wie folgt benannt werden können: „bereit zur Suche“, „Anzeige der Angebotsdaten“, „bereit zum Angebotanlegen“ und „bereit zur Angebotsänderung“. Jedem Zustand ist eine Maske zugeordnet.

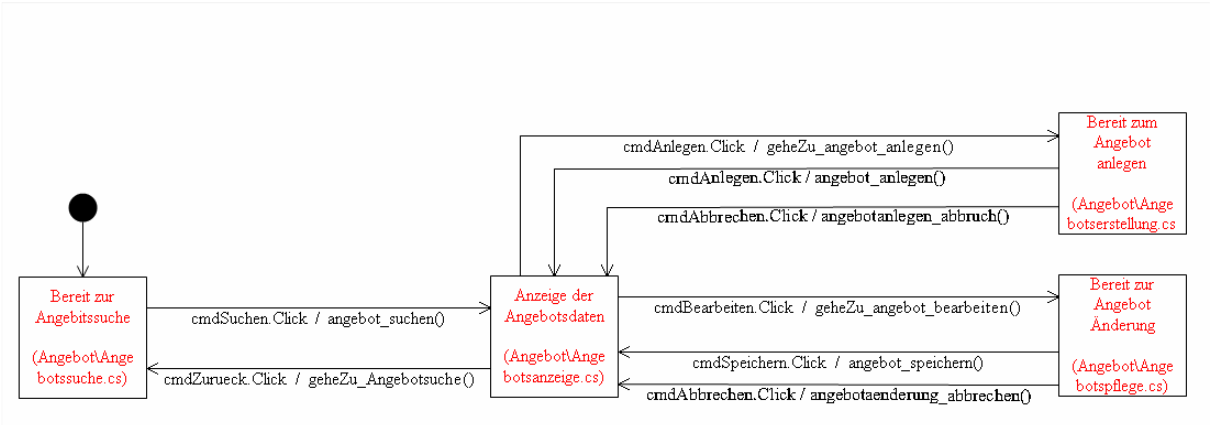


Abb. 36 Diagramm der Benutzungsschnittstelle des Angebotsverwaltungswerkzeugs

Das Zustandsdiagramm der Benutzungsschnittstelle des Angebotsverwaltungswerkzeugs in Abb. 36 wurde in SUIT erstellt. Jedem Zustand innerhalb des Zustandsdiagramms wurde eine Maske zugeordnet.

Zu Beginn befindet sich die Benutzungsschnittstelle im Zustand „bereit zur Suche“, um die zu bearbeitende Tupelmenge durch den Benutzer spezifizieren zu lassen.

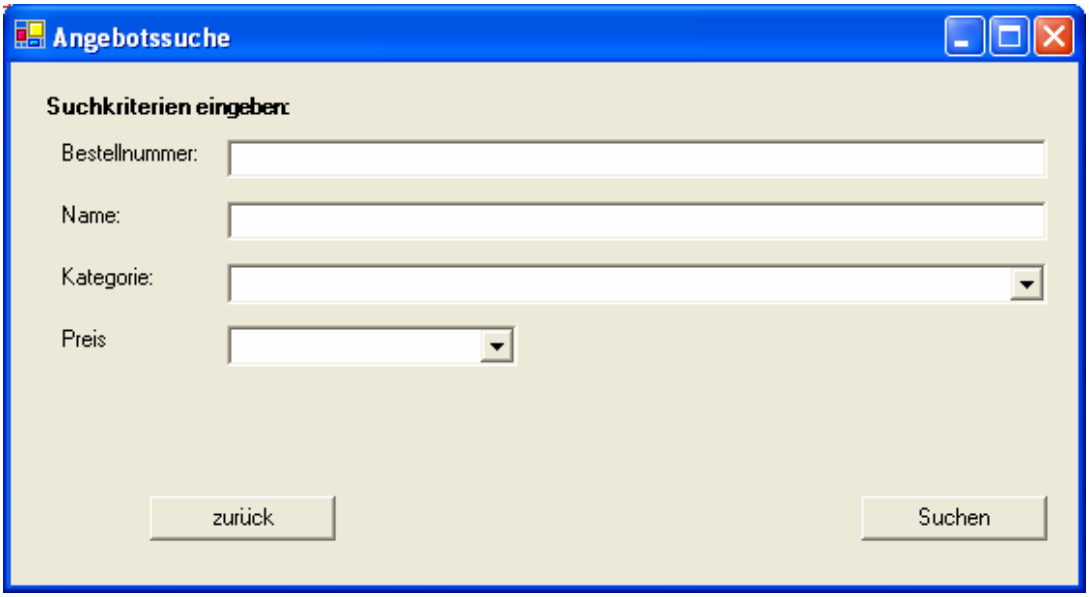


Abb. 37 Maske der Angebotsuche

Durch das Betätigen der Taste „suche“ in der Maske der Angebotssuche (Abb. 37) wird die Aktion Angebot_suchen() angestoßen. Die Benutzungsschnittstelle gelangt in den Folgezustand „Anzeige der Angebotsdaten“. Dieser ist der zentrale Zustand des Zustandsdiagramms, von dem aus die verschiedenen Operationen wie „Angebot ändern“, „Angebot löschen“ usw. ausgeführt werden können.

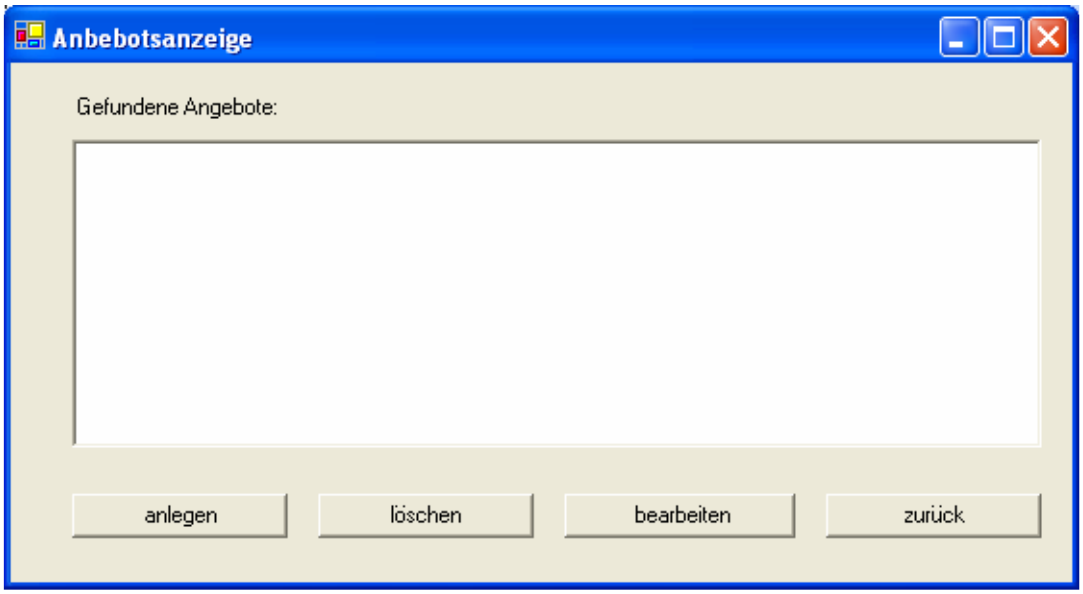


Abb. 38 Maske der Angebotanzeige

Durch das Betätigen der Taste „anlegen“ in der Maske der Angebotanzeige (Abb. 38) wird die Aktion „geheZu_angebot_anlegen()“ angestoßen. Die Benutzungsschnittstelle wechselt in den Zustand „bereit zum Angebotsanlegen“; hier sind nur noch der Abbruch oder das Erfassen möglich. Beide Tasten in der Maske Angebotserstellung in Abb. 39 stoßen jeweils eine Aktion an und führen zum Zustand „Anzeige der Angebotsdaten“ zurück.

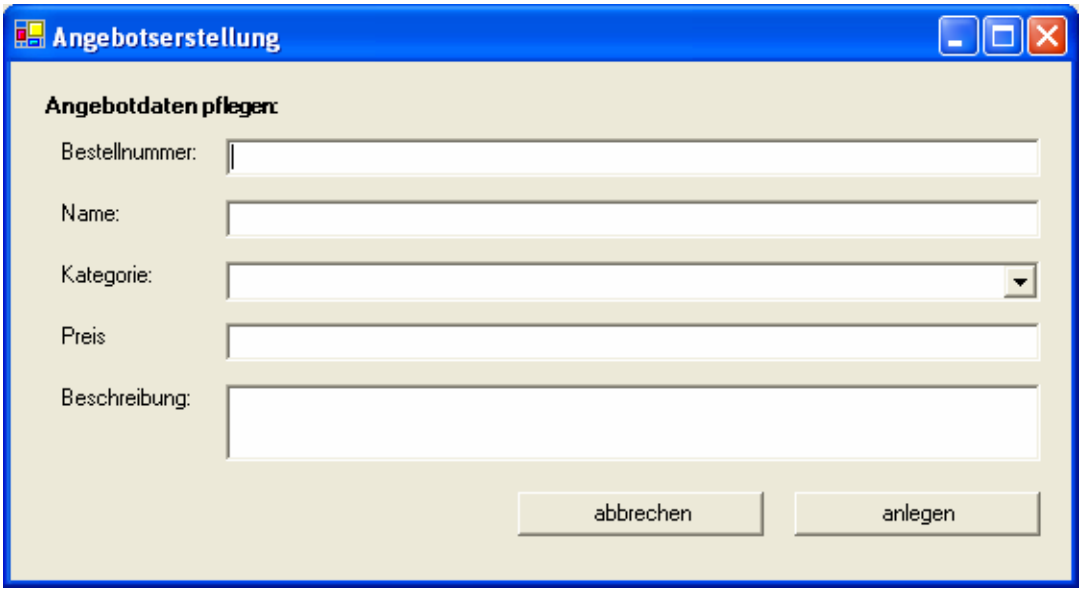


Abb. 39 Maske der Angebotserstellung

Vom Zustand „Anzeige der Angebotsdaten“ mit der Maske der Angebotsanzeige aus kann die Taste „bearbeiten“ betätigt werden. Dies führt zu der Aktion „geheZu_angebot_bearbeiten()“. Es wird zum Folgezustand „bereit zur Angebotsänderung“ gewechselt. In dieser Maske kann man über die beiden Tasten „abbrechen“ und „speichern“ und die dazugehörenden Aktionen in den Zustand „Anzeige der Angebotsdaten“ zurückkehren.

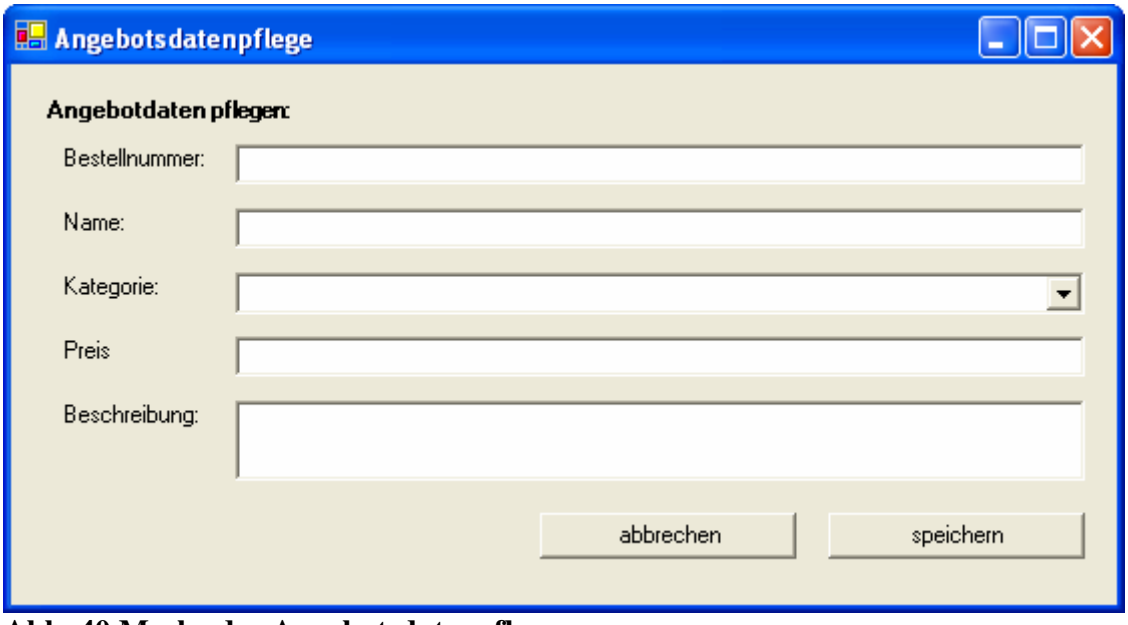


Abb. 40 Maske der Angebotsdatenpflege

Vom Zustand „Anzeige der Angebotsdaten“ aus kann man über die Taste „löschen“ nach Anklicken des zu löschenden Angebots oder durch Anklicken der Taste „zurück“ wieder in die Suchmaske zurückkehren, also zum Zustand „bereit zur Suche“.

7.3.5 Kundenverwaltungswerkzeug

Wie man dem Anwendungsfalldiagramm in Abb. 35 entnehmen kann, benutzen die Anwendungsfälle „Kunden anlegen“, „Kunden löschen“ und „Kundendaten pflegen“ den Anwendungsfall „Kunden suchen“. Daher muss eine Kundensuche ausgeführt werden, ehe eine weitere Aktion bezüglich eines Kunden erfolgen kann. Damit ist eine bestimmte Reihenfolge vorgegeben.

Wird dieser Ablauf durch ein Zustandsdiagramm beschrieben, ergeben sich daraus vier Zustände, die wie folgt benannt werden können: „bereit zur Suche“, „Anzeige der Kundendaten“, „bereit zum Kundenanlegen“ und „bereit zur Kundendatenpflege“. Jedem Zustand ist eine Maske zugeordnet.

Das Zustandsdiagramm inkl. der Zuordnung der Masken wird in Abb. 41 gezeigt.

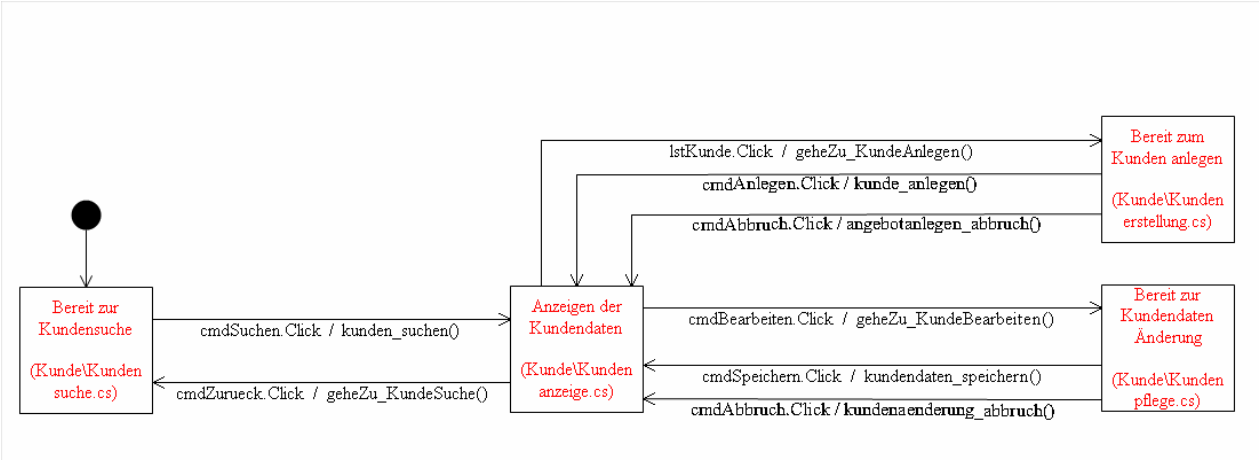


Abb. 41 Diagramm der Benutzungsschnittstelle des Kundenverwaltungswerkzeugs

Zu Beginn befindet sich die Benutzungsschnittstelle im Zustand „Bereit zur Kundensuche“, um die zu bearbeitenden Kundendaten durch den Benutzer spezifizieren zu lassen.

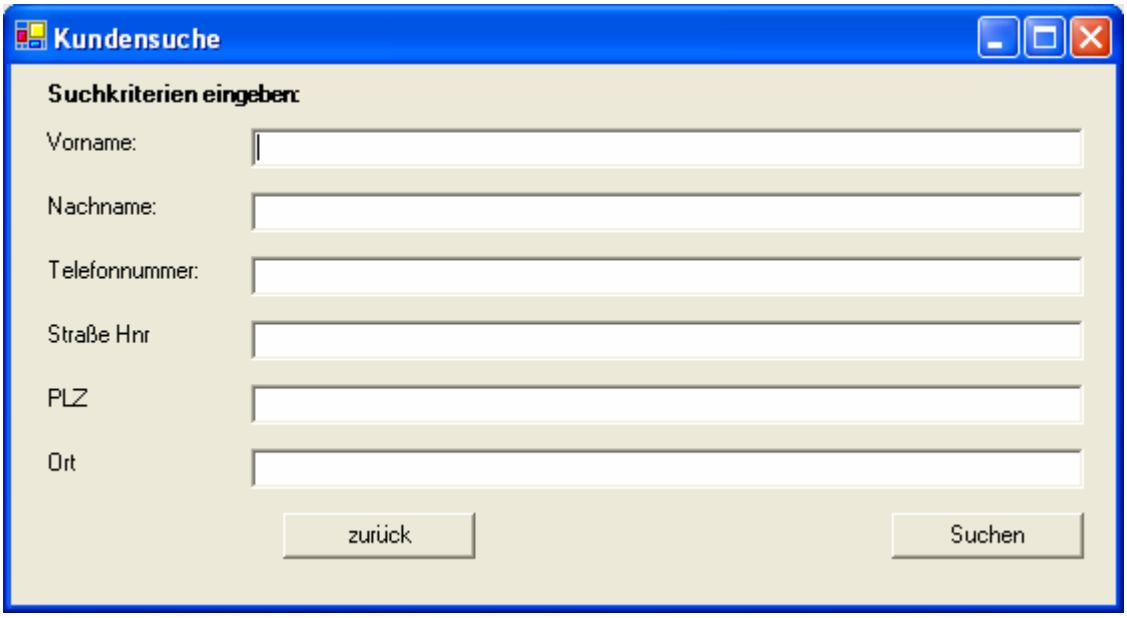


Abb. 42 Maske der Kundensuche

Durch das Betätigen der Taste „Suchen“ in der Maske der Kundensuche in Abb. 42 wird die Aktion geheZu_KundeSuchen() angestoßen. Die Benutzungsschnittstelle gelangt in den Folgezustand „Anzeige der Kundendaten“. Dieser ist der zentrale Zustand des Zustandsdiagramms, von dem aus die verschiedenen Operationen wie „Kunde anlegen“, „Kunde löschen“ usw. ausgeführt werden können.

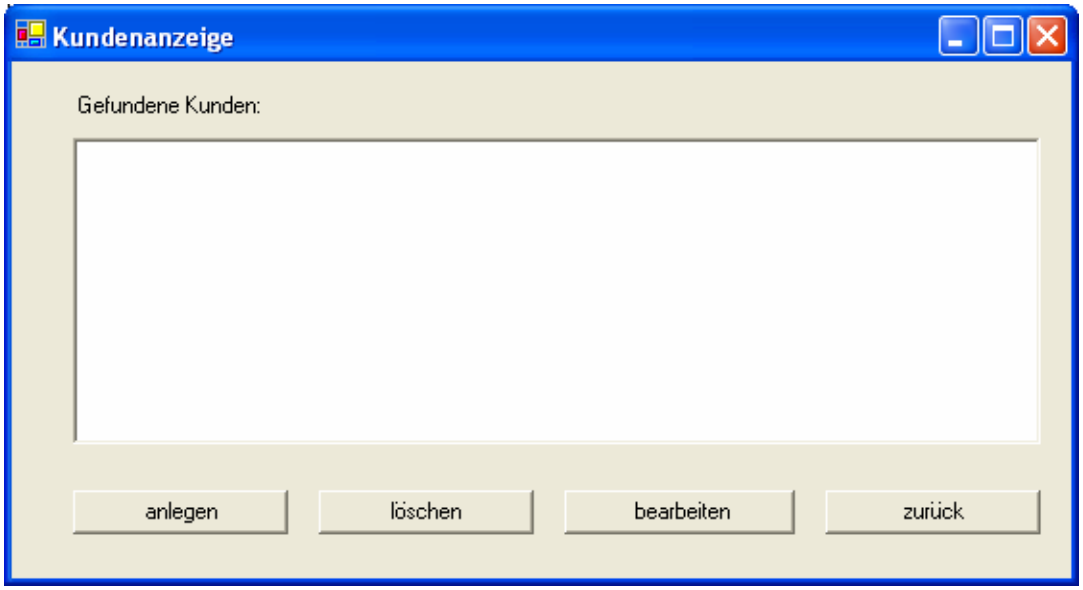


Abb. 43 Maske der Kundenanzeigen

Durch das Betätigen der Taste „anlegen“ in der Maske der Kundenerstellung in Abb. 44 wird die Aktion „geheZu_KundeAnlegen()“ angestoßen. Die Benutzungsschnittstelle wechselt in den Zustand „Bereit zum Kundenanlegen“; in der entsprechenden Maske sind nur noch der Abbruch oder das Erfassen möglich. Beide Tasten stoßen jeweils eine Aktion an und führen zum Zustand „Anzeige der Kundendaten“ zurück.

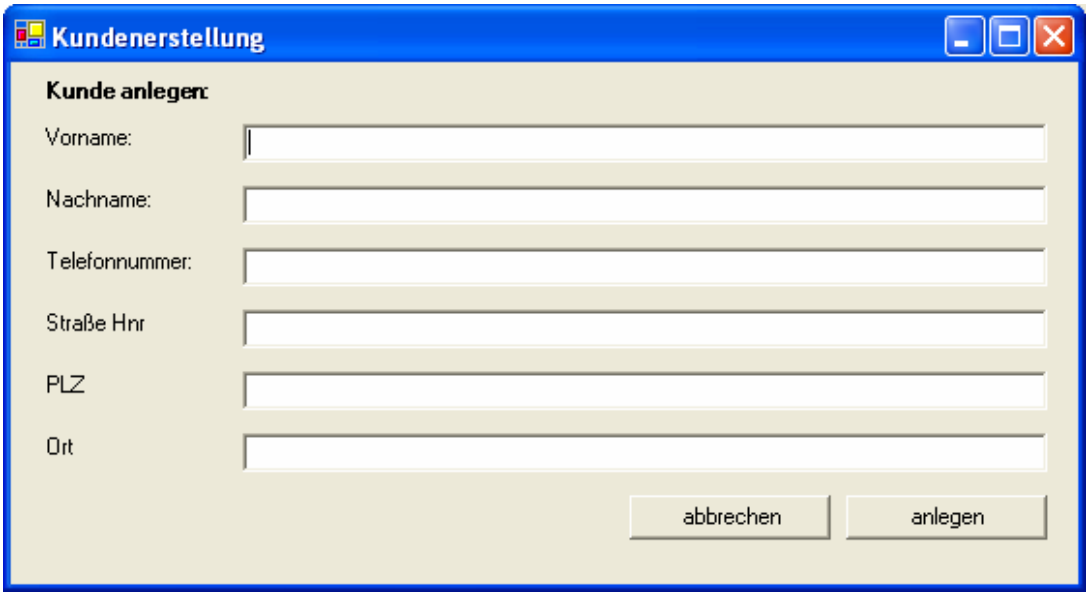


Abb. 44 Maske der Kundenerstellung

Vom Zustand „Anzeige der Kundendaten“ (mit der Maske in Abb. 43) aus kann die Taste „bearbeiten“ betätigt werden. Dies führt zu der Aktion „geheZu_KundeBearbeiten()“. Es wird zum Folgezustand „Bereit zur Kundendaten Änderung“ gewechselt. In der entsprechenden Maske (Abb. 45) kann man über die beiden Tasten „abbrechen“ und „speichern“ und die dazugehörenden Aktionen in den Zustand „Anzeige der Kundendaten“ zurückkehren.

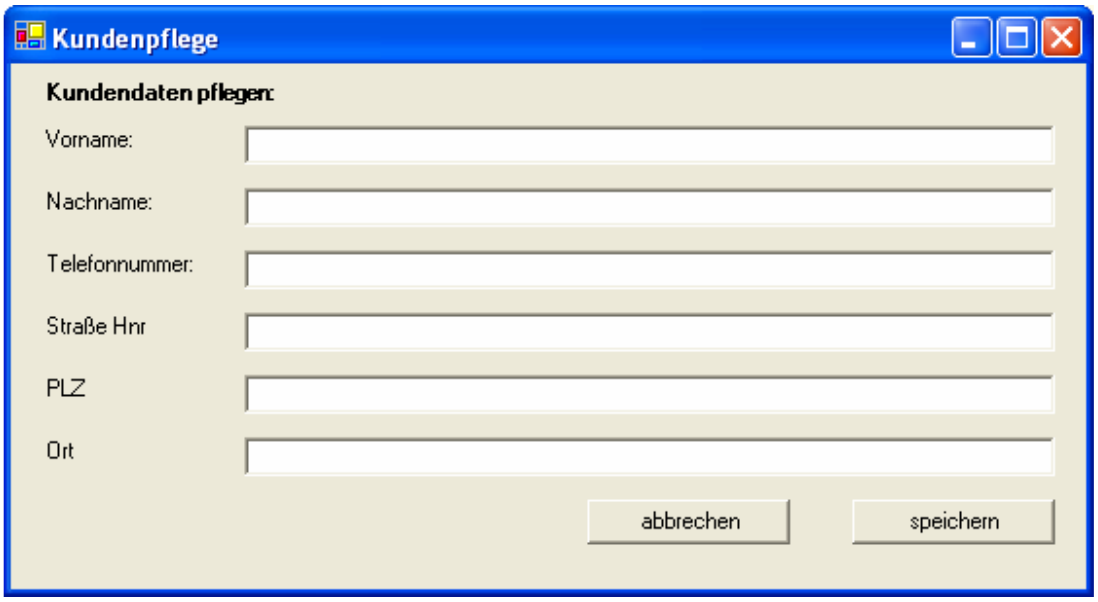


Abb. 45 Maske der Kundenpflege

Vom Zustand „Anzeige der Kundendaten“ bzw. der entsprechenden Maske aus kann man über die Taste „zurück“ wieder in die Suchmaske, also zum Zustand „bereit zur Suche“ zurückkehren.

7.3.6 Bestellverwaltungskombiwerkzeug

Wie dem Anwendungsfalldiagramm in Abb. 35 entnommen werden kann, nutzt der Anwendungsfall „Bestellung aufnehmen“ die fachlichen Dienstleistungen des Kundenverwaltungswerkzeug und des Angebotsverwaltungswerkzeug. Daher handelt es sich im Sinne von Züllighofen [Züllighoven 1998] um ein Kombi-Werkzeug, das zur Erledigung seiner Aufgabe verschiedene fachliche Dienstleistungen kombiniert und softwaretechnisch aus existierenden Subwerkzeugen zusammengesetzt ist.

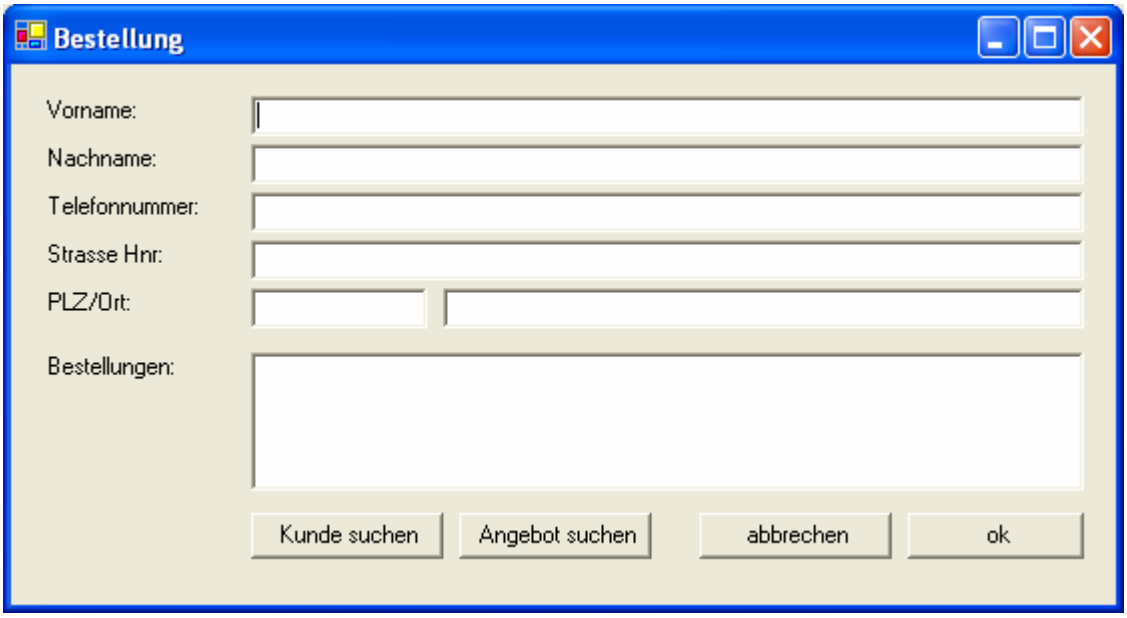


Abb. 46 Maske der Bestellung

Wird der Anwendungsfall „Bestellung aufnehmen“ durch ein Zustandsdiagramm beschrieben, ergibt sich der Zustand „Bereit zum Bestellen“ mit der entsprechenden Maske in Abb. 46, von welcher aus nach Betätigen der Taste „Kunde suchen“ in das Kundenverwaltungswerkzeug und durch Betätigen der Taste „Angebot suchen“ in das Angebotsverwaltungswerkzeug verzweigt werden kann .

Die Benutzungsschnittstellen der beiden Werkzeuge „Kundenverwaltungswerkzeug“ und „Angebotsverwaltungswerkzeug“ wurden bereits mit SUIT in Form von Zustandsdiagrammen und Fensterentwürfen modelliert (siehe Abb. 36 und Abb. 41).

Daher wäre es sinnvoll, diese Ergebnisse als „zusammengesetzte Zustände“, also Zustände, die die bereits modellierten Zustandsdiagramme repräsentieren, in das Zustandsdiagramm des Kombi-Werkzeugs „Bestellverwaltungskombiwerkzeug“ zu integrieren.

Die Verschachtelung von Zustandsdiagrammen und somit die Definition von zusammengesetzten Zuständen ist eine Anforderung, die wir erkannt haben. Sie trägt ganz entscheidend zur Reduzierung der Komplexität eines Modells bei und unterstützt den Werkzeuggedanken des WAM-Ansatzes. Die bereits identifizierten Werkzeuge werden im

Zustandsdiagramm des Gesamtsystems klar erkennbar, da sie als zusammengesetzte Zustände im Modell visualisiert werden.

Die Verschachtelung von Zustandsdiagrammen ist im Rahmen der vorliegenden Implementierung von SUIT noch nicht möglich. Die Realisierung dieser Anforderung wird im Rahmen der nächsten Version geplant.

7.3.7 Integration der Werkzeuge zum Gesamtsystem

Die Vorgehensweise der Untergliederung eines Gesamtsystems in Teilwerkzeuge hat sich, wie bereits erwähnt, in der Softwareentwicklung bewährt. Sie fördert die Verständlichkeit, verringert die Komplexität eines Systems und erhöht die Wiederverwendbarkeit.

Das Pizzaservicesystem wurde in die zwei einfachen (elementaren) Werkzeuge „Kundenverwaltungswerkzeug“ und „Angebotsverwaltungswerkzeug“ und in das Kombi-Werkzeug „Bestellverwaltungskombiwerkzeug“ untergliedert.

Diese Werkzeuge werden nun miteinander durch ein Zustandsdiagramm (Abb. 48) wieder zum Gesamtsystem integriert. Die Startmaske des Gesamtsystems Pizzaservice ist in Abb. 47 zu sehen.

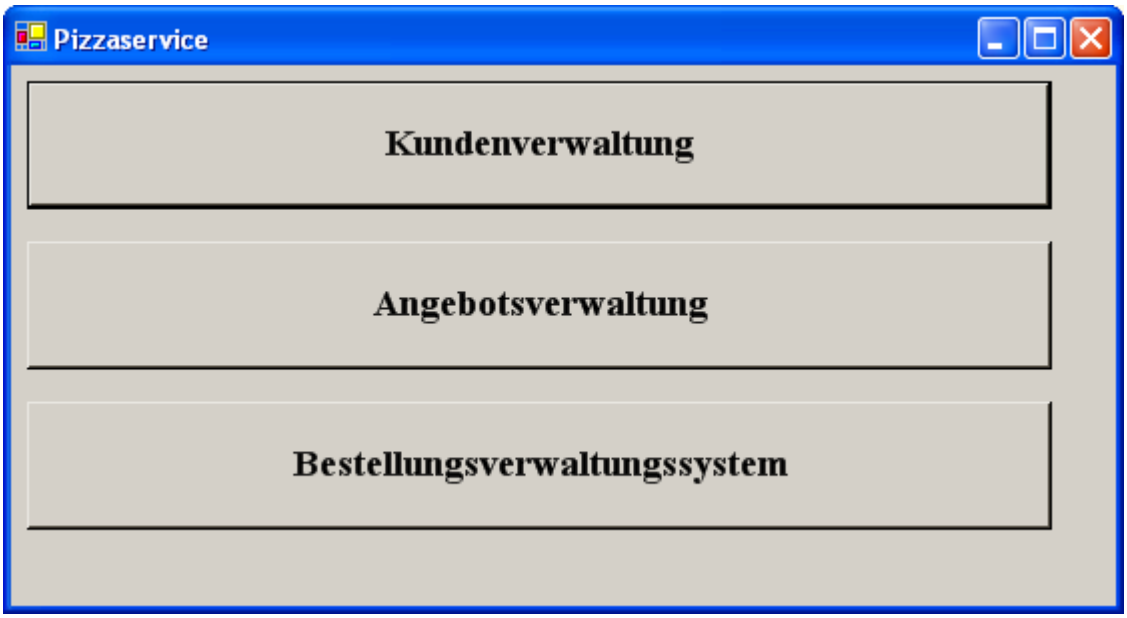


Abb. 47 Startmaske des Gesamtsystems Pizzaservice

Zu Beginn befindet sich das System im Zustand „start“. Durch das Betätigen der Taste „Kundenverwaltung“ gelangt das System in das Kundenverwaltungswerkzeug (Abb. 41), durch Betätigen der Taste „Angebotsverwaltung“ in das Angebotsverwaltungswerkzeug (Abb. 36). Über die Taste Bestellverwaltungssystem schließlich gelangt man in das Bestellverwaltungskombiwerkzeug (Abb. 48)

Mit Hilfe des entstandenen Zustandsdiagramms in Abb. 48 kann nun das Verhalten der Benutzungschnittstelle des Pizzaservicesystems simuliert werden. Das Zustandsdiagramm kann nun von der Simulationskomponenete interpretiert und automatisch ausgeführt werden.

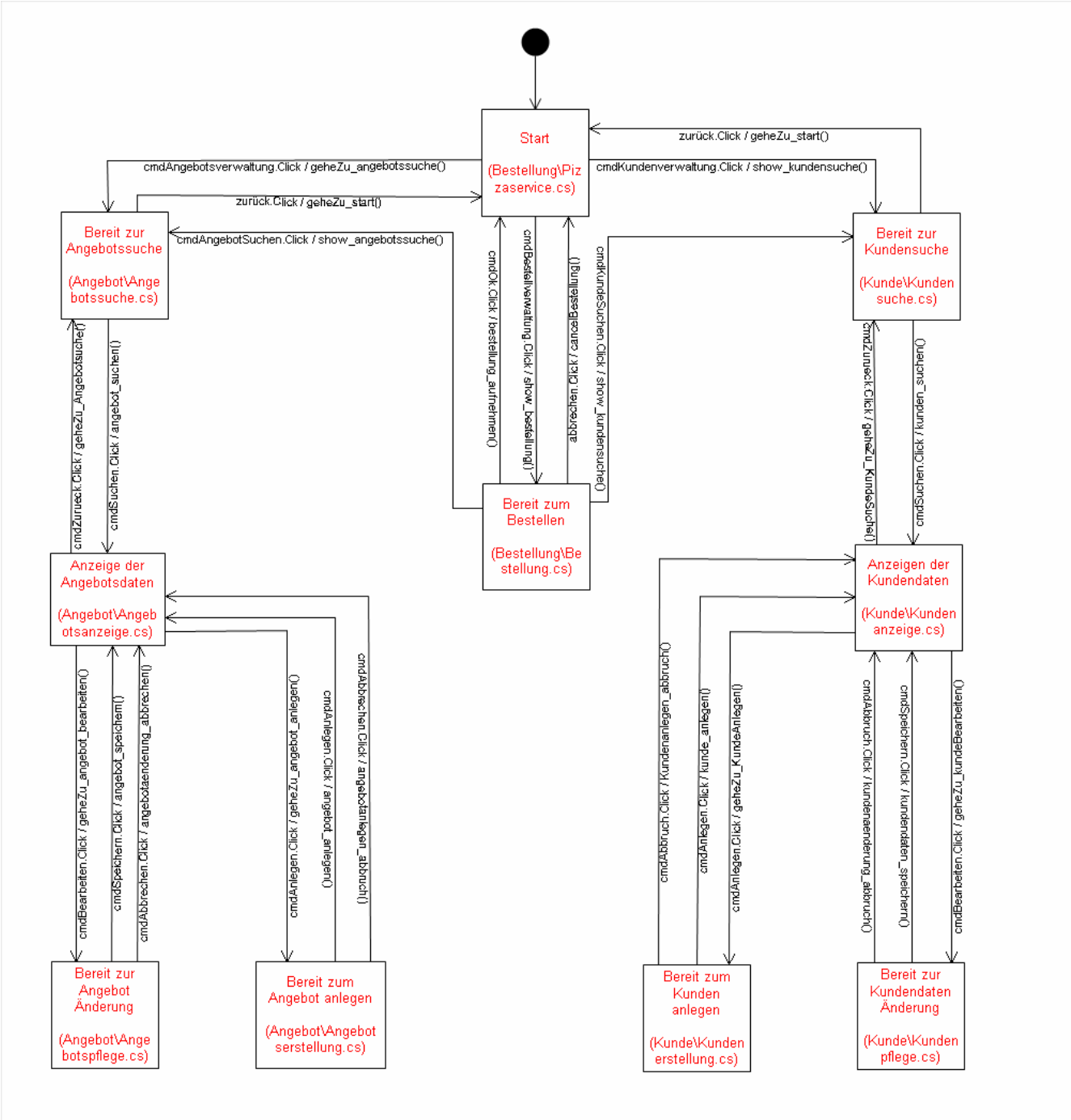


Abb. 48 Diagramm der Benutzungsschnittstelle des Gesamtsystems

SUIT erzeugt aus den im Zustandsdiagramm bereits definierten Aktionen eine Fassade. Für jede Aktion wird automatisch eine Methode generiert. Die fachliche Funktionalität dieser Aktionen muss vom Anwender nachträglich implementiert werden. Allein für das Zustandsdiagramm des Gesamtsystems Pizzaservice wurden 23 Methoden automatisch generiert. Bei großen Systemen ist die Menge der im Zustandsdiagramm definierten Aktionen entsprechend umfangreicher. Die automatische Generierung dieser Methoden in einer Fassade würde den manuellen Programmieraufwand erheblich reduzieren.

Folgender Fassaden-Ausschnitt wurde von SUIT für das Zustandsdiagramm in Abb. 48 erzeugt:

```
public class Fassade
{
      ...

    public static void angebot_bearbeiten(object sender, System.EventArgs e)
    {
        Output.WriteLine("angebot_bearbeiten() invoked.");
    }

    public static void angebot_suchen(object sender, System.EventArgs e)
    {
        Output.WriteLine("angebot_suchen() invoked.");
    }

    public static void show_angebotssuche(object sender, System.EventArgs e)
    {
        Output.WriteLine("show_angebotssuche() invoked.");
    }

    /// <summary>
    /// Code generated by FassadeGenerator. Don't change next statement.
    /// Change to this statement may cause exception if the code is regenerated.
    /// </summary>
    public static System.IO.TextWriter Output = System.Console.Out;
}
```

Abb. 49 Ein Ausschnitt der Fassade zum Zustandsdiagramm in Abb. 48

In der Simulationsansicht von SUIT kann man sich nun das bereits modellierte Verhalten des Pizzaservicesystems ansehen. Der aktive Zustand, in dem auf eine Benutzereingabe gewartet wird, wird in der Simulationsansicht farbig markiert. Erfolgt nun eine Benutzereingabe, so wird die Aktion zur besseren Nachvollziehbarkeit ausgegeben. Außerdem wird zum Folgezustand gewechselt.

Im Folgenden ist das Zustandsdiagramm des Pizzaservicesystems in der Simulationsansicht abgebildet:

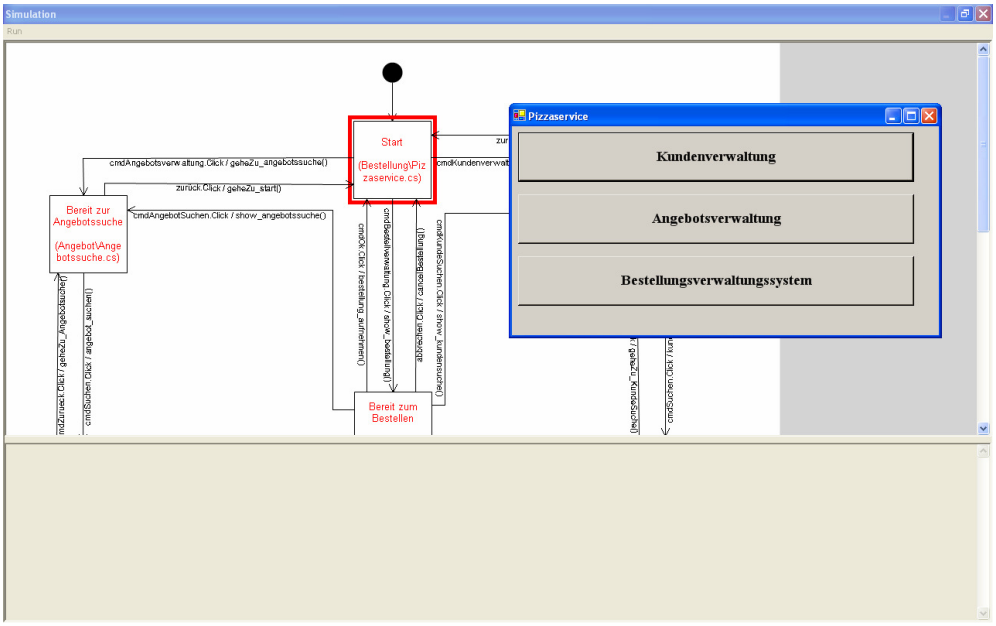


Abb. 50 Pizzasystem in der Simulationsansicht von SUIT

7.4 Fazit

Ziel von SUIT ist es, schnell und einfach einen Prototyp für eine Benutzungsschnittstelle zu erzeugen, um diese dem Anwender frühzeitig vorführen zu können. Außerdem soll der manuelle Programmieraufwand möglichst verringert werden. Aus den in SUIT erstellten Modellen in Form von Zustandsdiagrammen und Fensterentwürfen sollen so viele Informationen wie möglich zur automatischen Codegenerierung genutzt werden. Dieses Ziel wurde im Rahmen dieser Fallstudie am Beispiel eines Pizzaservicesystems exemplarisch erreicht.

In einem Zustandsdiagramm wurde das dynamische Verhalten des Pizzaservicesystems beschrieben. Hier wurden Zustände und Zustandsübergänge definiert. Jedem Zustand wurde eine Maske zugeordnet; die Zustandsübergänge wurden mit Beschriftungen für Ereignisse und Aktionen versehen.

Damit war der Prototyp fertig gestellt. Im Rahmen dieser Fallstudie hat SUIT am Beispiel des Pizzaservicesystems eine große Praxistauglichkeit bewiesen. Mit einem geringen Aufwand wurde aus dem Modell automatisch ein Prototyp erstellt. Darüber hinaus hat SUIT eine Fassade zur Weiterentwicklung automatisch generiert. Wenn also der Prototyp nicht als Wegwerfsoftware realisiert wird, kann er zu einem ausgereiften System weiterentwickelt werden.

Es kann festgehalten werden, dass SUIT mindesten für Systeme der Größenordnung des Pizzaservicesystems praxistauglich ist. Die Praxistauglichkeit bei komplexeren Systemen muss noch im Rahmen großer Projekte überprüft werden.

Die in diesem Kapitel verwendete Vorgehensweise hat, wie bereits erwähnt, primär das Ziel, zu demonstrieren, wie SUIT eingesetzt werden kann. Der Anwendungsbereich des Pizzaservicesystems wurde hierfür nur grob analysiert. Für einen Einsatz von SUIT für das Erzeugen von Benutzungsschnittstellen in der Praxis, ist eine detailliertere Anforderungsanalyse empfehlenswert, als im Rahmen der Fallstudie durchgeführt wurde. Auf diese Weise können Prototypen erfolgreich entwickelt werden, die fördernd in der Entwickler- und Benutzerkommunikation eingesetzt werden können. Sie könnten als Hilfsmittel dienen, um die Richtigkeit einer Anforderungsanalyse sicherzustellen. Anschließend können sie einem voll einsatzfähigen und kompletten Softwaresystem weiterentwickelt werden.

8. Usability

8.1. Einleitung

Thema dieses Kapitels ist der Usabiliy-Test, dem SUIT mit Hilfe von vier Testpersonen unterzogen wurde. Die Usability-Tests wurden nach zwei Hinsichten untersucht: 1. ging es um die Benutzbarkeit von SUIT selbst, 2. um die Praxistauglichkeit der Methode, die SUIT umsetzt (das Modellieren von Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen).

8.2 Usability Aufgabe

Jeder Testperson wurde eine Aufgabe gestellt, die sie störungsfrei bearbeiten sollte. Die Aufgabe bestand darin, anhand eines Beispiels (Suche nach Literatur, siehe nächster Abschnitt) in SUIT ein Zustandsdiagramm zu entwerfen, das den beschriebenen Ablauf der Literatursuche modelliert und dieses um die benötigten Masken anreichert. Es sollte eine Fassaden-Klasse erzeugt und das Modell simuliert werden.

Beispiel Suche nach Literatur:

Martina hat sich dazu entschieden, ihr Referat über das Thema "Geschichte der Dinosaurier" anzufertigen, und ist nun auf Literatursuche. Sie begibt sich deshalb in die Bibliothek, setzt sich an einen Rechner und öffnet die Seite des Katalogs der Bibliothek. Daraufhin öffnet das System den Suchdialog mit Eingabefeldern für die möglichen Suchkriterien. Da Martina sich erst einmal einen grundsätzlichen Überblick über ihr Themengebiet verschaffen möchte, gibt sie im Eingabefeld "suchen" den Text "Dinosaurier" ein und klickt mit der Maus den Button "suche" an. Kurze Zeit später präsentiert das System die Ergebnisse in einer Liste mit mehreren Einträgen. Neben interessanten Titeln wie „australische Dinosaurier“ finden sich darin auch weniger relevante Titel. Martina klickt auf den Titel eines für ihr Referat nützlich erscheinenden Buches, notiert sich die Signatur und geht zu den Regalen hinüber.

8.3 Testverlauf

Als Kriterien für die Auswahl der Testpersonen legten wir fest, dass sie Informatik oder technische Informatik studieren und die Vorlesung Software Engineering besuchen oder

besucht haben. Auf diese Weise sollten nur Testpersonen ausgewählt werden, die Kenntnisse in der Modellierung und in Prototyping mitbringen.

Vor dem Usability-Test fand ein Vorgespräch statt: Anhand eines kleinen, mit SUIT gestalteten Beispiels wurde die Testperson mit der Methode, Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen zu modellieren, und mit der Bedienung von SUIT vertraut gemacht.. Anschließend erhielt die Testperson die Aufgabe, die sie selbständig zu lösen hatte. Nach jedem Test fand dann ein Nachgespräch statt, in dem die Testperson einzeln über die im Folgenden genannten Punkte befragt wurde. Das Nachgespräch lässt sich in drei Hauptthemen gliedern: Vorkenntnisse, Bedienung und Methode.

Folgende Fragen wurden der Testperson gestellt:

Vorkenntnisse:

1. Haben Sie schon einmal mit Visual Studio.Net gearbeitet?

2. Haben Sie schon einmal mit einem Zustandsdiagramm gearbeitet? Wenn ja, mit welchem? Und wie würden Sie SUIT anhand Ihrer Erfahrungen im Vergleich mit den anderen Zustandsdiagrammeditoren beurteilen?

3. Haben Sie schon einmal Prototypen für BSS erstellt? Wenn ja, wie umfangreich war der investierte Aufwand?

Bedienung:

4. Wie beurteilen Sie die Bedienung von SUIT? Sehen Sie vor allem Schwächen oder Stärken in der Bedienung von SUIT?

Methode:

5. Wie beurteilen Sie die Idee (den Ansatz), BSS mit Hilfe von Zustandsdiagrammen und Fensterentwürfen zu entwerfen? Fällt Ihnen diese Denkweise schwer?

6. Würde die Genierung einer Fassade für Sie eine Arbeitserleichterung bedeuten?

7. Stellt die Simulation für Sie eine hilfreiche Veranschaulichung ihres Modells dar?

8. Würden Sie SUIT für die Erstellung ihrer nächsten Prototypen nutzen wollen?

9. Welche. Schwachstellen sehen Sie in SUIT, welche Verbesserungsvorschläge würden Sie anführen?

8.4 Ergebnisse

Der Usability-Test fand in einem Usability-Labor [uselab] der Hochschule für Angewandte Wissenschaften Hamburg statt. Alle von der Testperson vorgenommenen Eingaben über

Tastatur und Maus wurden während der Bearbeitung der Testaufgabe in SUIT aufgezeichnet. Neben dem Bildschirmgeschehen wird auch das Verhalten der Testperson aufgezeichnet: Zu diesem Zweck sind im Usability-Labor mehrere Kameras aufgestellt. Während der Aufnahme kann die Testperson durch lautes Denken ihre Eindrücke bei der Benutzung von SUIT schildern, während sie die Aufgabe löst.

Im Usability-Labor werden diese Schilderungen durch Ton und Videoaufnahmen aufgezeichnet. Aus den spontanen Äußerungen der Testperson kann man bei der Auswertung Schlüsse darüber ziehen, wie die Testperson mit SUIT zurechtgekommen ist. Außerdem findet nach jedem Test ein Nachgespräch statt, in dem die Testperson ihre Erfahrungen mit SUIT schildern und Verbesserungsvorschläge äußern kann. Insbesondere kann sie Auskunft darüber geben, an welchen Stellen sie Schwierigkeiten bei der Benutzung hatte.

All diese Beobachtungen und Äußerungen fließen in die Auswertung des Usability Tests ein.

8.4.1 Erste Testperson

Die Testperson hat noch keine Erfahrung mit der Entwicklungsumgebung Visual Studio. Sie hat bereits einmal Zustandsdiagramme erstellt, jedoch nicht im Zusammenhang mit Benutzungsschnittstellen und nicht in der Kombination von Zustandsdiagrammeditor und GUI-Builder, wie sie in SUIT vorliegt.

Bisher hat ihr eine Methode oder Vorgehensweise gefehlt, mit der sie schnell und einfach Prototypen erstellen kann. Die Kombination von GUI-Builder und Zustandsdiagrammeditor und somit die Möglichkeit, Zuständen Masken zuzuordnen, ist ihr zugänglich und erscheint ihr sehr intuitiv in der Umsetzung. Die ihr bekannten Tools zum Erstellen von Zustandsdiagrammen wie z.B. Together sind aus ihrer Sicht mit z.T. unnötigen Funktionalitäten überladen und, anders als der Zustandsdiagrammeditor von SUIT, unübersichtlich und komplex. Die Einführung in SUIT gestaltet sich daher einfach. SUIT ist außerdem anschaulich, so dass man sich, wie die Testperson hervorhebt, auf die „eigentliche Aufgabe konzentrieren kann“

Ein weiterer Punkt ist, dass SUIT, anders als Together, auf die Modellierung von BSS abzielt. Die Vorteile von SUIT liegen für die Testperson in der drastischen Vereinfachung der Modellierung des dynamischen Verhaltens einer Benutzungsschnittstelle, die durch den Einsatz der aufeinander abgestimmten Tools Zustandsdiagrammeditor und GUI-Builder zustande kommt. Damit ist die Benutzungsfreundlichkeit von SUIT ein entscheidender Vorteil.

Weiterhin ist für die Testperson wichtig, dass SUIT frühzeitig ein simulierbares Ergebnis bietet. Die Simulation erlaubt, schnell Ergebnisse zur Verfügung zu haben. Besonders vorteilhaft findet die Testperson die automatische Codegenerierung, da sie den Programmieraufwand erheblich reduziert.

Die Testperson würde SUIT generell benutzen, da dieses nach ihrer Ansicht „eine schnelle und einfache Denke“ ermöglicht. Außerdem findet die Testperson die Möglichkeit der Simulation in verschiedenen Ansichten praktisch. Der Entwickler könne, so schätzt die Testperson die Möglichkeiten ein, die Simulation in der Ansicht des Zustandsdiagramms

ablaufen lassen und die Dialogabläufe anhand des von ihm modellierten Zustandsdiagramms nachvollziehen. Dem Kunden hingegen, der in der Regel weniger an Details interessiert sei, könnte daher die Simulation in Form einer bloßen Abfolge von Masken, so wie sie ihn später als Software vorliegt, präsentiert werden.

In der gesamten Video- und Tonaufzeichnung (auch der Formulierung ihrer Eindrücke) wirkt die Testperson entspannt; sie erledigte die Testaufgabe sicher und ohne Schwierigkeiten. Insgesamt hat die Testperson zweiundzwanzig Minuten für die Lösung der Usability Aufgabe benötigt (siehe CD:\\Usability\test01.avi).

Im Folgenden wird in Abb. 51 das Ergebnis der ersten Testperson aufgezeigt [CD:\\Usability\ Test-1\ Test-1.sln]:

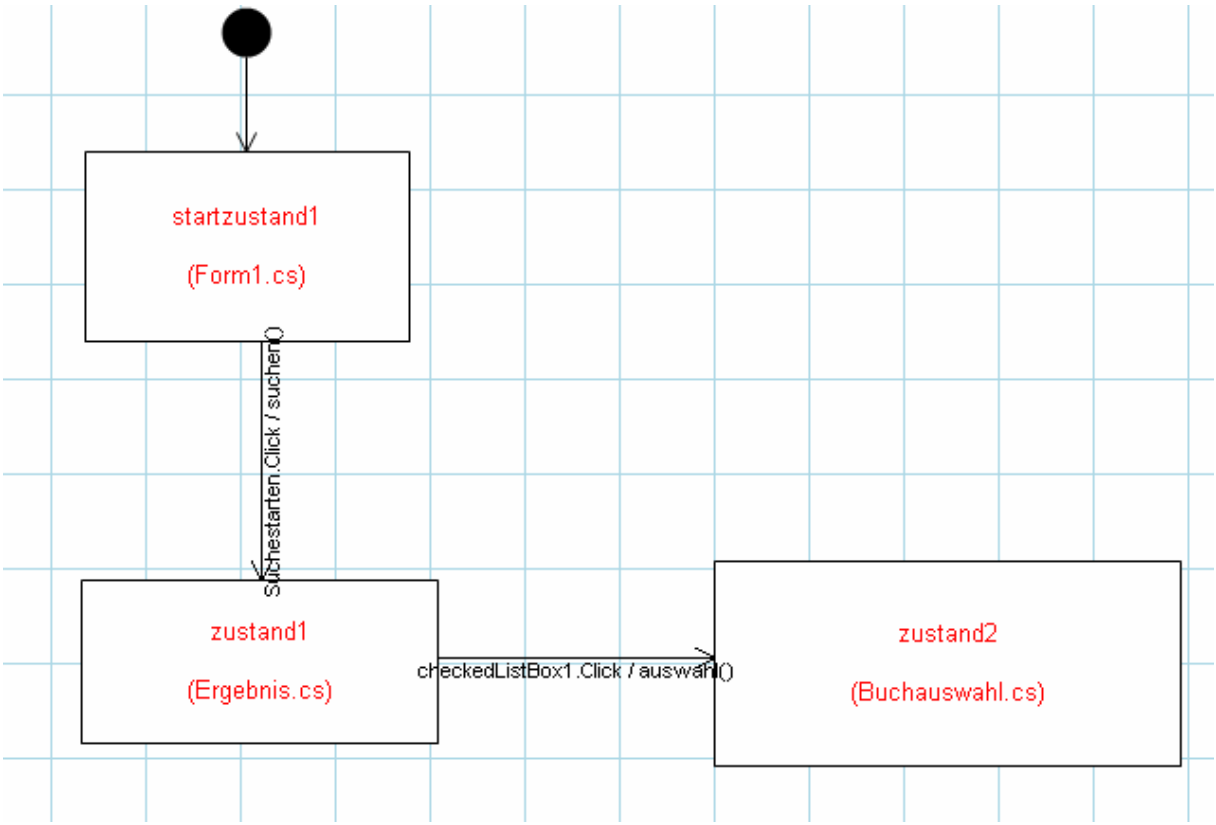


Abb. 51 Lösung der ersten Testperson

8.4.2 Zweite Testperson

Die Testperson hat noch nicht mit der Entwicklungsumgebung Visual Studio gearbeitet. Sie ist dieser gegenüber reserviert und abgeneigt, sie zu benutzen. Als in hohem Maße störend empfindet sie die Toolbox in Visual Studio, die den Designer verdeckt.

Die Testperson bringt Erfahrung in der Zustandsdiagrammmodellierung mit; sie hat auch bereits einmal Prototypen erstellt. Vor dem Hintergrund dieser Erfahrung geht die Erstellung eines Prototyps in SUIT aus ihrer Sicht sehr schnell und einfach vonstatten. Die Erstellung

von Prototypen in Form von Zustandsdiagrammen und Fensterentwürfen fiel der Testperson nicht schwer.

Die Testperson bewertet es sehr positiv, dass in der Fassade einprogrammierter Code durch eine Änderung im Zustandsdiagramm nicht verloren geht. Die Bedienung in SUIT weist aus Sicht der Testperson keine Mängel auf. Wäre SUIT nicht in Visual Studio integriert, würde sie das Tool gerne benutzen. Für die Lösung der Usability Aufgabe hat die Testperson dreizehn Minuten gebraucht (siehe CD:\\Usability\test02.avi).

So sieht das Ergebnis der Testperson aus [CD:\\Usability\Test-2\Test-2.sln]:

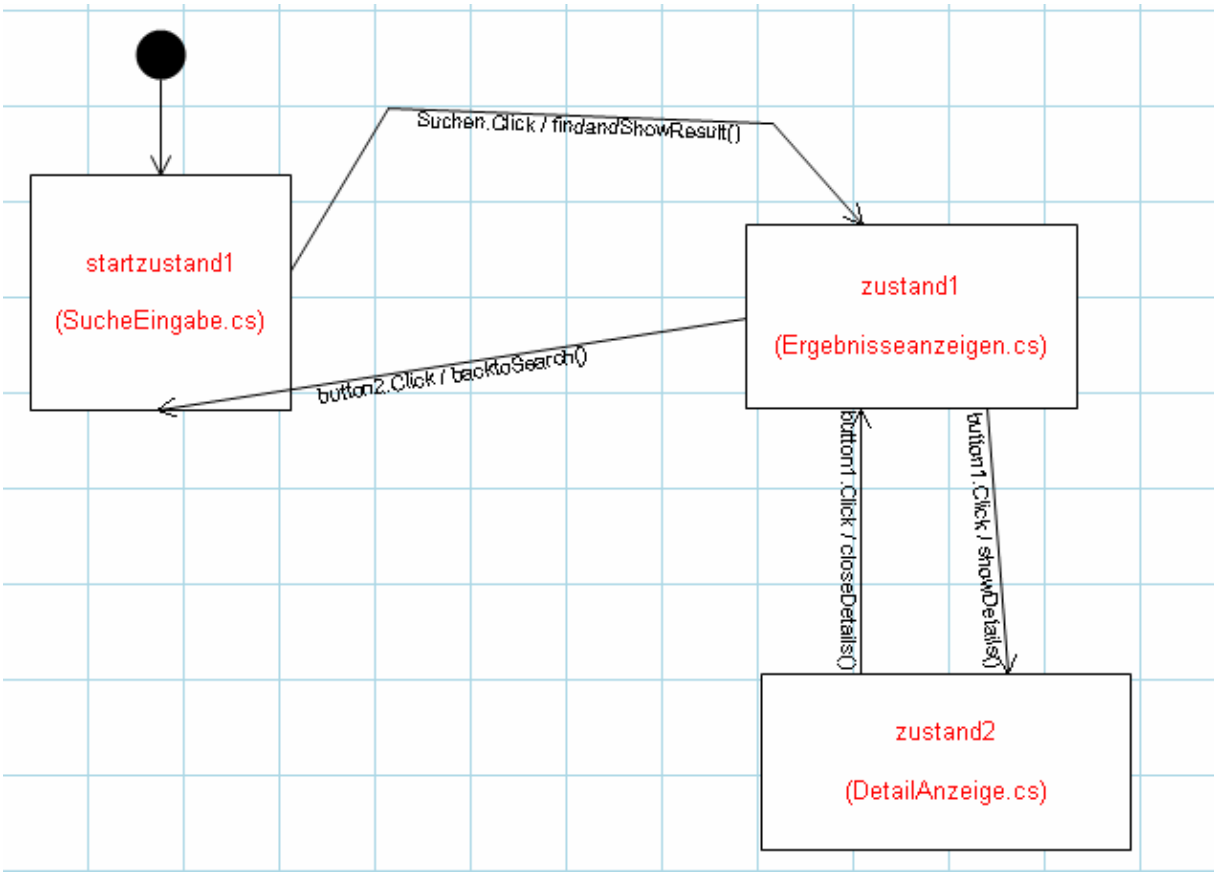


Abb. 52 Lösung der zweiten Testperson

8.4.3 Dritte Testperson

Die Testperson bringt keine Erfahrung in Visual Studio mit. Sie beurteilt die Umgebung als gewöhnungsbedürftig und zu überladen. Für die Erstellung von Zustandsdiagrammen hat die Testperson bisher Together oder Poseidon benutzt. Außerdem besitzt sie Erfahrung mit der Entwicklung von Prototypen; sie hat diese allerdings immer manuell erstellt, was im Verhältnis zu SUIT mehr Programmieraufwand erfordert hat. Nach Aussage der Testperson werde mit SUIT ein „einfaches Design“ ermöglicht. Die Simulation biete ihr eine „gute Hilfe“. Die Modellierung in Form von Zustandsdiagrammen und Masken ist aus Sicht der Testperson verständlich und zugänglich. Die automatische Codegenerierung der Fassade sieht

die Testperson als positiv an. Die Bedienung in SUIT ist für sie einfach, bedarf aber aufgrund der Entwicklungsumgebung Visual Studio einer Einarbeitung.

Insgesamt findet die Testperson SUIT „viel versprechend“ und würde dieses Tool gerne benutzen wollen, wäre es nicht in Visual Studio realisiert. Insgesamt hat die Testperson neunzehn Minuten für die Überarbeitung der Usability Aufgabe gebraucht (siehe CD:\\Usability\ test03.avi).

So sieht das Ergebnis der von der Testperson gelösten Aufgabe aus (Abb. 53) [CD:\\Usability\ Test-3\ Test-3.sln]:

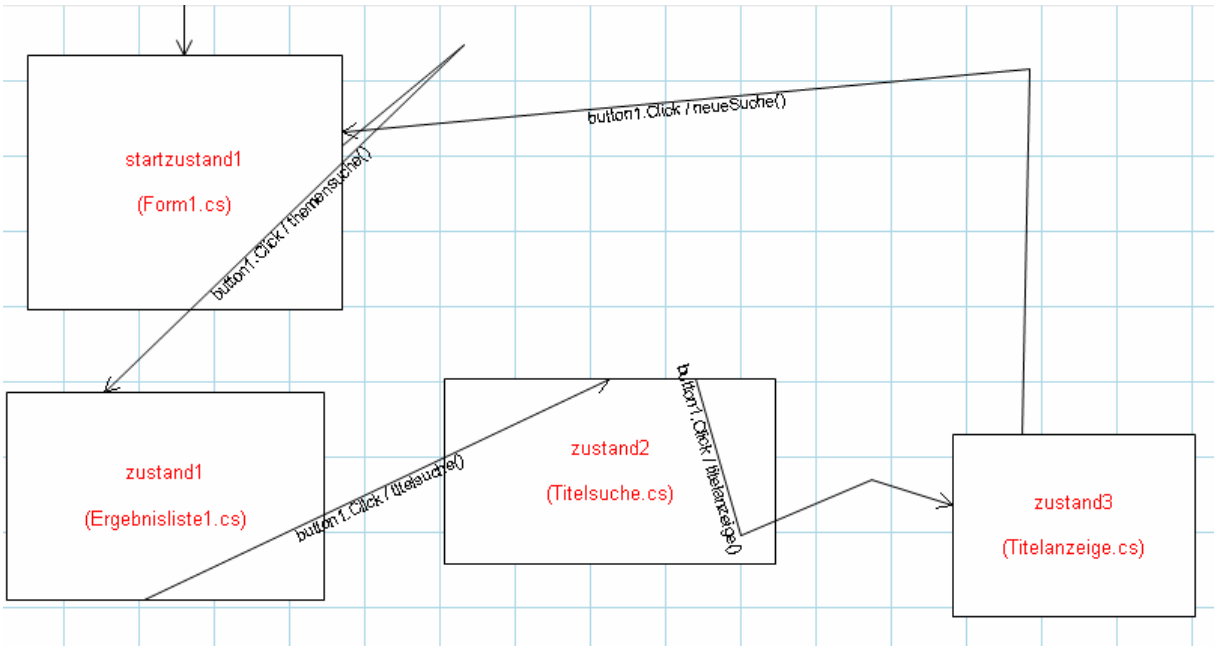


Abb. 53 Lösung der dritten Testperson

8.4.4 Vierte Testperson

Die Testperson bringt keine Erfahrung in Visual Studio mit; sie beurteilt die Entwicklungsumgebung jedoch nicht negativ. Bisher hat die Testperson Together für die Erstellung von Zustandsdiagrammen benutzt und findet im Vergleich dazu den Zustandsdiagrammeditor in SUIT übersichtlich und anschaulich. Die Testperson hat noch keine Prototypen erstellt, vermutet aber, dass dies einen großen Programmieraufwand mit sich bringt, wenn die Erstellung manuell erfolgt.

Mit SUIT seien diese dagegen „schnell gemacht“. Die Testperson hat sich bei der Aufgabe schnell zurechtgefunden. Die Testperson findet, dass die Simulation eine gute Veranschaulichung für den Kunden bietet. Für sie bietet SUIT eine wesentliche Erleichterung in der Prototypenerstellung, da in SUIT Änderungen schnell im Zustandsdiagramm vorgenommen und simuliert werden können.

Die Testperson äußert aber auch, dass sich möglicherweise bei der Modellierung größerer Systeme die Modellierung in Form von Zustandsdiagrammen als problematisch erweisen könnte.

Bezüglich der Bedienung von SUIT hat die Testperson keine Verbesserungswünsche geäußert, jedoch weiß sie, dass die Arbeit in SUIT die Einarbeitung in Visual Studio voraussetzt.

Die Testperson hat siebzehn Minuten für die Bearbeitung der Usability Aufgabe gebraucht (siehe CD:\\Usability\test04.avi). So sieht das Ergebnis der vierten Testperson aus [CD:\\Usability\ Test-4\Test-4.sln]:

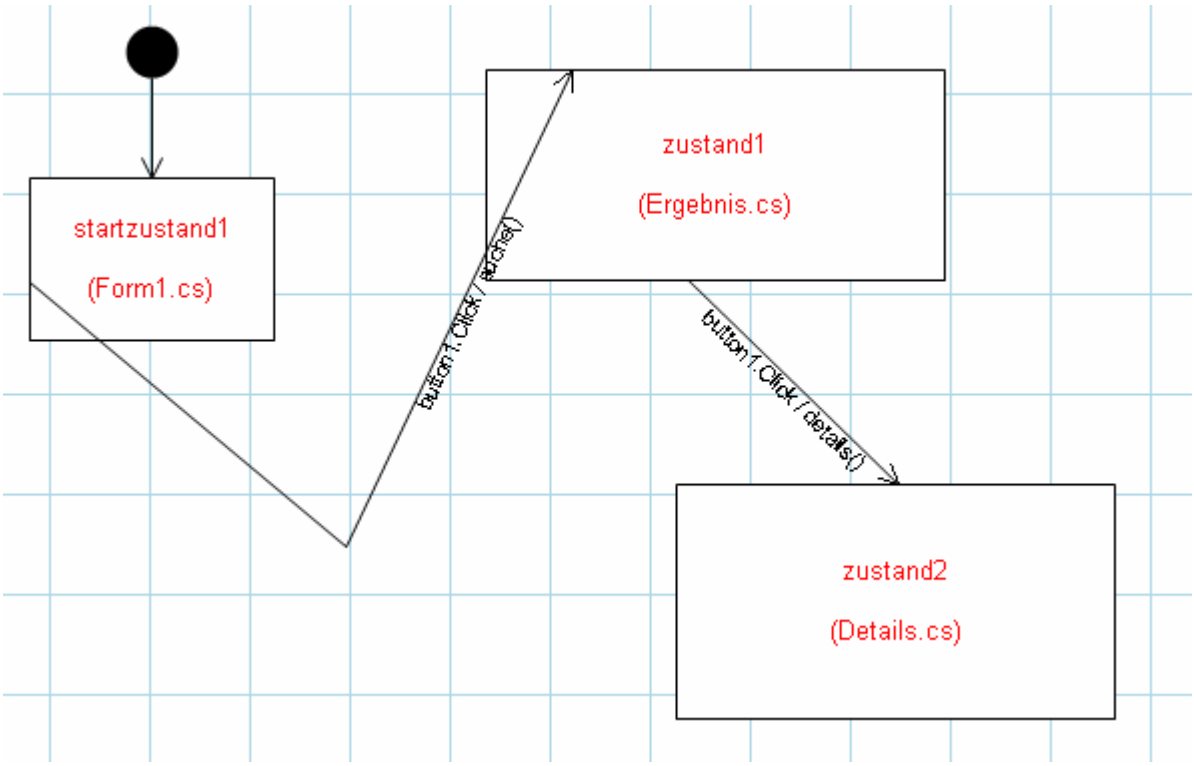


Abb. 54 Lösung der vierten Testperson

8.5 Fazit

Insgesamt hat keine Testperson mehr als fünfundzwanzig Minuten für die Erledigung der Usability- Aufgabe gebraucht. Das ist für uns aus mehreren Gründen ein positives Ergebnis: Erstens wurden die Testpersonen mit einem neuen Tool und einer ihnen bisher unbekannten Methode (der Modellierung von Benutzungsschnittstellen mit Hilfe von Zustandsdiagrammen und GUI-Buildern) konfrontiert; zweitens besaß keine Testperson Erfahrung mit Visual Studio, (der Entwicklungsumgebung, in die SUIT integriert ist).

Der Usability-Test hat ergeben, dass die Methode, Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen zu modellieren, anschaulich und intuitiv ist.

Diese Denkweise ist keiner der vier Testpersonen schwer gefallen. Alle Testpersonen haben die Aufgabe sicher erledigt. Die Erstellung von Prototypen in Form eines Modells, dessen Ergebnis simulierbar ist, beurteilen alle sehr positiv, da dies keinerlei Programmieraufwand erfordert. Die Fassade bietet für alle Testpersonen einen weiteren Vorteil, da sie den zusätzlichen Programmieraufwand reduziert.

Zwei der Testpersonen besaßen keine Erfahrung in der Prototyperstellung. Gründe dafür sind die bisher fehlenden Methoden und Werkzeuge, mit denen eine schnelle und einfache Prototyperstellung möglich wäre. Die anderen zwei Testpersonen besaßen Erfahrung in der Erzeugung von Prototypen, beschrieben diese jedoch aufgrund der manuellen Vorgehensweise als aufwändig und unpraktisch. Alle Testpersonen äußerten sich positiv über die von SUIT umgesetzte Methode und würden mit SUIT gerne weitere Erfahrungen machen.

Bei zwei Testpersonen stellte die Entwicklungsumgebung Visual Studio ein Argument gegen die Benutzung von SUIT dar, da beide das Arbeiten mit Visual Studio eher als störend empfinden. Ansonsten beurteilten alle die Bedienung von SUIT als ergonomisch und benutzerfreundlich. Insgesamt ergab der Test, dass SUIT eine Bereicherung für die Prototypentwicklung darstellt. Dies deutet darauf hin, dass es dafür bisher an geeigneten Tools mangelt.

9. Möglichkeiten und Grenzen von SUIT

9.1 Einleitung

In diesem Kapitel soll auf die Möglichkeiten und Grenzen von SUIT eingegangen werden, die im Laufe der Bachelor-Arbeit aufgefallen sind. Ob SUIT die erhofften Vorteile in der Vision erfüllt oder nicht wird in Abschnitt 9.2 erläutert. Diese Grenzen von SUIT sollen in Abschnitt 9.3 aufgezeigt und ihre Ursachen erläutert werden. In Abschnitt 9.4 folgt ein Fazit zu diesem Kapiteln.

9.2 Möglichkeiten von SUIT

Die Methode, Benutzungsschnittstellen mit Hilfe von Zustandsdiagrammen und Fensterentwürfen zu entwerfen, wurde mit SUIT umgesetzt. Erste praktische Tests haben ergeben, dass SUIT eine gute Basis bietet, Benutzungsschnittstellen schnell und kostengünstig zu entwerfen. Die Usability-Tests zu dieser Methodik Benutzungsschnittstellen zu entwerfen, hatten gezeigt, dass alle Tester diese Vorgehensweise als leicht verständlich betrachten und ohne großes Umdenken Zustandsdiagrammmodelle ihres Benutzungsschnittstellensystem intuitiv entwerfen könnten (siehe Kapitel 8).

Die Simulationskomponente kann schon in den frühen Phasen der Softwareentwicklung zum Einsatz kommen. Sie ermöglicht, ohne eine Zeile Code geschrieben zu haben, einen Prototyp der zu entwickelnden Benutzungsschnittstelle zu erzeugen Die Simulationskomponente bietet die Möglichkeit des Rapid-Prototyping und stellt damit eine Bereichung für die Benutzer-Entwickler-Kommunikation dar. Prototypen können erstellt werden, von Benutzern getestet und falls Verbesserungswünsche existieren, entsprechend geändert werden. Nur wenn die Erstellung von Prototypen schnell genug ist, sind mehrere Prototypingzyklen möglich, in denen der Prototyp iterativ verbessert wird. Sind z.B. Änderungen erforderlich, so kann das modellierte Zustandsdiagramm in SUIT angepasst werden und schon hat man einen neuen Prototyp, den man dem Kunden vorzeigen kann.

Die Fassaden-Genierung hingegen nimmt dem Entwickler Programmieraufwand ab, und kann damit dazu beitragen, Kosten und Zeit zu sparen. Die Fassaden-Klasse stellt eine Schnittstelle zwischen der Funktions- und Interaktionskomponente dar und bietet somit eine Möglichkeit, den erzeugten Prototypen weiter zu entwickeln, so dass der Prototyp nicht als Wegwerfprototyp angesehen werden muss. Die Fassaden-Genierung ist so angelegt, dass bei Änderungen im Zustandsdiagramm-Modell und erneuten Genierung der Fassade, bereits in der Fassaden-Klasse von Anwender hinzuimplementierter Code nicht verloren geht.

So bietet SUIT nicht nur ein Tool zum schnellen Erzeugen von explorativen Prototypen von Benutzungsschnittstellen, sondern diese Prototypen sollen und können sogar weiter entwickelt

werden. Diese Möglichkeit ist ihm gegeben, da SUIT sich in Visual Studio.Net integriert und somit der Entwickler schon in seiner Entwicklungsumgebung ist.

Die Vorteile, die wir in der Vision beschrieben haben, scheinen sich also mit SUIT erfüllt zu haben. SUIT erfüllt also in der Hinsicht unsere Erwartungen. Jedoch haben wir auch Grenzen an SUIT entdeckt, die in Abschnitt 9.3 genauer erläutert werden. Unter anderem ist eine dieser Grenzen von SUIT das Entwickeln von verteilten Benutzungsschnittstellen (siehe 9.3.1). So beschränkt sich SUIT erstmal nur auf Einzelplatzsystemen.

Ein kleines Einplatzsystem für einen Pizzaservice (siehe Fallstudie) wurde mit SUIT ansatzweise entwickelt. Anhand der Fallstudie hat sich SUIT für solche Benutzungsschnittstellen als gut geeignet erwiesen. Wie tauglich SUIT für die Entwicklung von größeren Benutzungsschnittstellen ist, lässt sich erst genau sagen, wenn SUIT wirklich zum praktischen Einsatz kommt.

9.3 Erläuterung der Grenzen

In diesem Abschnitt werden die erkannten Grenzen von SUIT im Laufe der Bachelor-Arbeit erläutert und aufgezeigt. Es sollen die Ursachen festgestellt werden und mögliche Lösungen angeboten werden. Diese Lösungen – sofern realisierbar – können dann für die nächste Version von SUIT umgesetzt werden.

9.3.1 Verteilte Benutzungsschnittstellen

Mit SUIT lässt sich nicht der Dialogablauf von Benutzungsschnittstellen modellieren, deren Bildschirmmasken sich auf mehreren Rechnern befinden. Dies hat als Ursache, dass der SUIT Zustandsdiagramm-Editor keine projektübergreifende, geschweige denn rechnerübergreifende Referenzierung der Bildschirmmasken erlaubt. In der vorliegenden SUIT-Version ist es noch nicht möglich, auf ein externes Visual Studio Projekt zuzugreifen, selbst wenn das externe Projekt sich auf demselben Rechner befinden würde. So ist ein verteiltes Benutzungsschnittstellensystem nicht mit SUIT zu modellieren, bzw. nicht ohne dass der Programmierer selber Hand anlegt und die externen Bildschirmmasken programmiertechnisch seinem Projekt zugänglich macht. Diese projektübergreifende Bildschirmmasken-Referenzierung der Zustände im SUIT Zustandsdiagramm-Editor ist für die nächste Version geplant.

Durch diese Einschränkung der Referenzierung ist es daher auch nicht möglich, dass auf Web-Benutzungsschnittstellen zugegriffen werden kann, da zurzeit auch noch keine Referenzierung von Web-Seiten erlaubt ist. Um dies zu ermöglichen, müsste, neben einer GUI-Builder Komponente, SUIT um einen Web-Seiten-Editor erweitert werden. Von dieser Komponente wurde in dieser SUIT-Version abgesehen, da eine Untersuchung der zustandsbasierten Modellierung von Benutzungsschnittstellen auch ohne die Web-Seiten-Editor Komponente möglich ist, und diese Untersuchung primäres Ziel dieser Bachelor-Arbeit ist.

9.3.2 Codegenerierung für den Aufruf des Folgezustands

Mit dem SUIT-Zustandsdiagrammeditor wird der Dialogverlauf einer Benutzungsschnittstelle modelliert. Die Informationen, welche Maske bei bestimmten Ereignissen angezeigt wird, sind also in dem Zustandsdiagrammmodel enthalten. Es ist also möglich, diese Informationen aus dem Model zu entnehmen, um die Codegenerierung für den Folgezustand, also das Anzeigen der Folgemaske, zu automatisieren. Jedoch sind beim Generieren dieser Codestellen, die für das Aufrufen deren Folgemaske zuständig sein werden, einige Dinge zu beachten. Wo soll der Code generiert werden? In der Transition angebenden Aktion, also in der dazugehörigen Aktionsmethodendeklarationen der Fassade, oder doch lieber in der Maske-Klasse selbst? Egal wo der Code hingeneriert wird, beide Stellen hätten ihre Vor- und Nachteile.

Um diese Vor- und Nachteile zu erläutern, soll folgendes Beispiel genutzt werden. Abb. 55 zeigt ein SUIT Zustandsdiagramm, das das dynamische Verhalten einer kleinen, noch unvollständigen Benutzungsschnittstelle beschreiben soll.

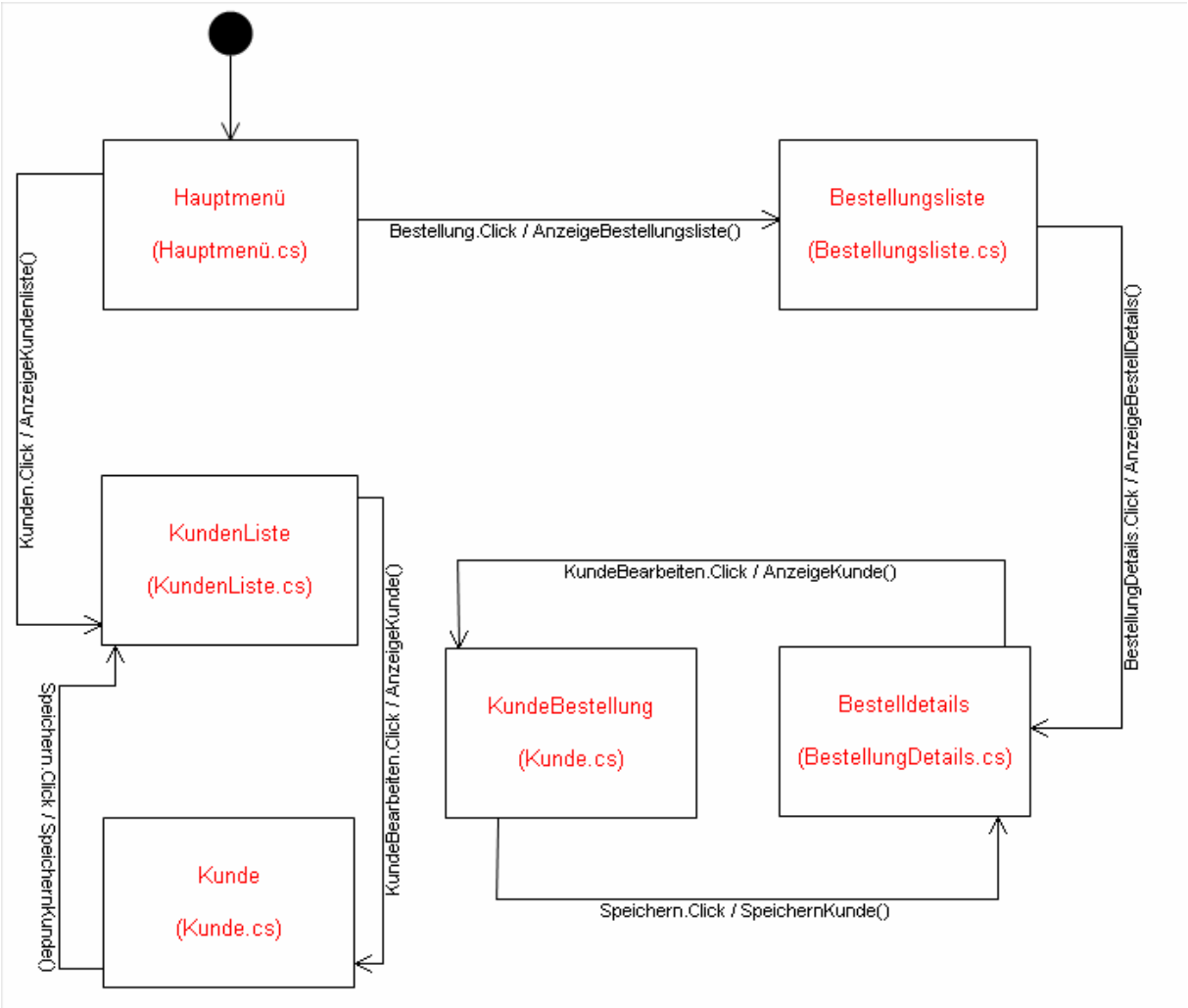


Abb. 55 SUIT-Zustandsdiagrammbeispiel

In Abb. 55 wird eine Benutzungsschnittstelle modelliert, die mit Kundendaten arbeitet. Diese Kunden geben Bestellungen aus, die auch von dieser Benutzungsschnittstelle verwaltet

werden sollen. Über das Hauptmenü, in dem sich das System zu Anfang befindet, kann der Anwender entscheiden, ob er Kunden- oder Bestelldaten ansehen will. Je nachdem, was der Anwender ansehen will, bekommt er eine Liste mit der Übersicht alle Kunden- oder Bestelldaten. In diesen Übersichtslisten kann der Anwender dann die Details der Daten zu einem Kunden – sofern er vorher im Hauptmenü die Kundenliste ausgewählt hat - oder zu einer Bestellung anfordern und manipulieren.

Zu einer detaillierten Beschreibung einer Bestellung gehört ein Kunde, der diese Bestellung aufgegeben hat. Dieser Kunde kann im Bestelldetail angezeigt und bearbeitet werden. Daher ist die Kunden-Maske (***Kunde.cs***) sowohl über den Weg der Kundenliste als auch über die Bestellliste erreichbar. Diese Kunden-Maske wird, wie in Abb. 55 zu erkennen, in zwei Zuständen (***Kunde*** und ***KundeBestellung***) benutzt. Eine Wiederreferenzierung von Masken in unterschiedlichen Zuständen ist in manchen Fällen also sinnvoll und wird von SUIT unterstützt. Diese unterstützte Funktionalität macht jedoch die Codegenerierung für den Aufruf des Folgezustands – und somit die Folgemaske – komplexer als sie ist.

Das Problem bei der Codegenerierung des Folgemaskeaufrufs besteht darin, zu entscheiden, an welcher Stelle dieser Code generiert werden muss. Es gibt dort drei mögliche Ansätze, die auf den ersten Blick alle ihre Berechtigung haben:

A. Code für den Aufruf der Folgemaske wird in der Aktion selbst generiert

B. Code für den Aufruf der Folgemaske wird in der vorigen Maske generiert

C. Zur Laufzeit wird mit Hilfe des Zustandsdiagrammmodels bestimmt, welche Folgemaske aufgerufen wird

Für die Generierung nach Möglichkeit A sprich folgendes: in Abb. 55 sind einige Aktionen mit den Namen ***AnzeigeKundenliste***(), ***AnzeigeBestellungsliste***(), ***AnzeigeBestelldetails***() und ***AnzeigeKunde***() zu sehen. Diese Aktionen sind von den Namen her so gewählt, dass der Schluss nahe liegt, diese Aktionen sorgen für den Aufruf der entsprechenden Folgemaske, wie z.B. die Kundenliste oder die Bestellliste. Der Code für den Folgemaskeaufruf sollte also bei diesen Aktionen idealerweise in der Aktion selbst generiert werden.

Nur im Umkehrschluss sind solche Funktionen eigentlich überflüssig, wenn sie als Funktionalität nur den Aufruf der Folgemasken haben. Diese Informationen sind im Modell schon in den Transitionen durch deren Folgezustand enthalten, und müssen nicht durch eine entsprechend benannte Aktion zusätzlich in das Modell eingebaut werden. Da jedoch in der aktuellen SUIT Zustandsdiagramm-Notation zu jeder Transition ein Ereignis und eine Aktion angegeben werden muss, sind ***AnzeigeKundeliste***(), ***AnzeigeBestellungsliste***(), ***AnzeigeBestellungdetails***() und ***AnzeigeKunde***() notwendig, damit eine dieser Transitionen angegeben werden kann. Die Notwendigkeit dieser Methode könnte außerdem noch darin liegen, dass z.B. AnzeigeKunde() als Funktionalität hätte, dafür zu sorgen, dass der ausgesuchte Kunde aus der Datenbank geholt und angezeigt werden muss. Jedoch ist das nicht immer der Fall, dass beim Zustandsübergang eine Aktion benötigt wird. Diesen Mangel soll in der nächsten SUIT Version behoben werden. Es ist geplant, dass die SUIT Zustandsdiagrammnotation für die nächste Version auf die Eigenschaft hin erweitert werden kann, Aktionen bei Transitionen als eine optionale Angabe anzusehen und nicht mehr als notwendige Pflichtangabe.

Wenn man unter diesem Aspekt genauer überlegt, ist es fragwürdig, ob der Code für den Aufruf der Folgemaske wirklich idealerweise immer in der Aktion generiert werden sollte. Dagegen spricht vor allem die Tatsache, dass es nicht in jeder Aktion Sinn macht, den Code für den Folgemaskeaufruf zu generieren. Da es Aktionen geben kann, nach deren Ausführung unterschiedliche Folgemasken geladen werden müssen, ist eine Codegenerierung für den Aufruf der Folgemaske in der Aktion sogar fatal. In Abbildung 4 ist eine Aktion ***SpeichernKunde***() zu erkennen, die in zwei Transitionen benutzt werden. Diese zwei Transitionen haben unterschiedliche Folgemasken, einmal zur ***BestellungDetails.cs*** und einmal zu ***KundenListe.cs***. Würde hier Code für den Aufruf der Folgemaske in der Aktion ***SpeichernKunde***() generiert werden, so ist völlig unklar, ob nur die Folgemaske nun ***BestellungDetails.cs*** oder ***KundenListe.cs*** sein muss.

Wenn also der Code nicht allgemein in der Aktion geniert werden kann, so wäre es als zweite Möglichkeit die Codegenerierung nach Möglichkeit B zu untersuchen. Möglichkeit B geniert den Code für den Aufruf der Folgemaske in der vorigen Maske.

Im Beispiel der Abb. 55 würde das bedeuten, dass in der Maske ***Hauptmenü.cs*** selbst der Code hinterlegt wird, der dafür sorgt, dass beim Klicken auf *Bestellung* die Maske ***Bestellungsliste.cs*** und beim Klicken auf *Kunden* die Maske ***Kundenliste.cs*** aufgerufen werden. In der Maske ***Hauptmenü.cs*** den Code für den Aufruf zu erzeugen scheint also in diesem Fall gut zu funktionieren.

Funktioniert dies auch in den meisten Fällen, so gibt es auch hier wieder Masken, wo die Codeerstellung des Aufrufs der Folgemaske nicht in der Maske selbst erzeugt werden kann. In Abb. 55 wird die Maske ***Kunden.cs*** in zwei Zuständen verwendet. Würde nun also der Code der Folgemaske in der Maske ***Kunde.cs*** hinterlegt werden, so wäre nicht klar, was die Folgemaske sein wird, im Falle das *Speichern* geklickt würde. Speichern.Click führt bei dem Zustand ***KundeBestellung*** zurück zu ***BestellDetails.cs*** während beim Zustand ***Kunde*** zu ***KundenListe.cs*** gegangen werden muss. Der Code für den Aufruf der Folgemaske würde also bei ***Kunden.cs*** dazu führen, dass für Speichern.Klick keine klare Folgemaske definiert werden kann, ohne zu berücksichtigen in welchem Zustand wir uns gerade befinden.

Dieses Problem kann dadurch vermieden werden, dass die Doppelreferenzierung von Masken verboten wird. Will man also eine Maske in einem anderen Zustand wieder verwenden, müsste man eine Kopie dieser Maske erzeugen und die Kopie dann referenzieren. Jedoch führt diese Vorgehensweise zu redundanten Masken, die Nachteile in der Wartung und Änderungen dieser Masken mit sich bringen.

SUIT verzichtet aus diesem Grunde auch auf die Erzeugung des Codes für den Aufruf des Folgezustands in der vorigen Maske selbst.

Da SUIT weder den Code in der Aktion noch in der vorigen Maske erzeugt, muss ein anderer Weg gefunden werden. Da bei einer Doppelrefenzierung von Masken in verschieden Zuständen es dazu kommen kann, dass eine Maske nur in Zusammenhang mit dem Wissen über ihren aktuellen Zustand bestimmen kann, was ihre Folgemaske ist, ist dieser Weg also nur unter Einbeziehung des aktuellen Zustands, und somit dem Zustandsdiagrammmodell möglich.

Der aktuelle Zustand, in dem ein Benutzungssystem sich gerade befindet, lässt sich nur zur Laufzeit bestimmen. Die dritte Lösung wäre also, zur Laufzeit über eine zusätzliche

Komponente zu bestimmen, was gerade der aktuelle Zustand ist, und was für ein Zustand, somit auch welche Maske, beim Auslösen des Ereignisses folgen muss.

Diese Komponente arbeitet Hand in Hand mit dem Zustandsdiagramm zur Laufzeit und müsste also zusätzliche Ressourcen beanspruchen, die auf Kosten der Ressourcen für das Benutzungsschnittstellensystem gehen. Außerdem würde eine Komponente, die Teil der Benutzungsschnittstelle werden müsste, für den SUIT-Anwender bedeuten, dass er eine zusätzliche Komponente in seinem Projekt zu verwalten und verstehen hat.

Aus diesem Grunde wurde in SUIT auch auf diese Komponente verzichtet. In der aktuellen SUIT Version wird somit keinerlei Codegenierung für den Aufruf des Folgezustands unterstützt. SUIT überlässt dies der Zuständigkeit des Programmierers, da dieser am besten bestimmen kann, wie und wo diese Codestellen hingehören.

In der aktuellen SUIT Version wird bei der Simulation mit Hilfe des Zustandsdiagrammmodels bestimmt, welcher Zustand der Aktuelle ist, um dann den entsprechenden Folgezustand / Folgemaske aufzurufen. Aber für das endgültige Benutzungsschnittstellesystem muss der SUIT-Anwender selbst dafür sorgen.

9.2.6 Konsistenzverlust in späteren Phasen

Mit der jetzigen SUIT Zustandsdiagramm-Notation haben wir erreicht, dass Benutzungsschnittstellen schnell und verständlich mit einem Zustandsdiagramm modelliert werden können. Die Usabilty-Tests haben gezeigt, dass diese unkomplizierte SUIT Zustandsdiagramm-Notation stark dazu beigetragen hat, diese Methodik der Benutzungsschnittstellen-Modellierung verständlich und intuitiv umsetzbar zu halten. Ziel ist es, ein Tool zu bieten, das möglichst früh in den Etappen der Softwareentwicklung zum Einsatz kommt. Das Abbilden von funktionalen Elementen war daher nie bestreben der SUIT Zustandsdiagramm-Notation. Jedoch lässt sich eine Benutzungsschnittstelle in ihrer endgültigen Version niemals komplett ohne die Informationen über die Funktionalität, wie z.B. Rückgabewerte einzelner Funktionen, beschreiben, die durchaus Einfluss auf den Dialogverlauf nehmen können.

Für das SUIT Zustandsdiagramm-Modell könnten diese fehlenden funktionalen Elemente in den späteren Etappen der Softwareentwicklung bedeuten, dass es seine Konsistenz verliert, da eben nicht der komplette Dialogverlauf einer Benutzungsschnittstelle unter Berücksichtigung der funktionalen Implementierung im Modell selbst abbildbar ist. So kann zwar der mit SUIT entstandene Prototyp zum Weiterentwickeln genutzt werden, jedoch ist es wahrscheinlich, dass das SUIT Zustandsdiagramm-Modell später seine Gültigkeit verliert.

Diese Konsistenzerhaltung, und die damit verbunden SUIT Zustandsdiagramm-Notation Erweiterungen könnten als primäres Ziel für die folgenden SUIT Versionen angesehen werden. Diese möglichen Erweiterungen und deren Ursachen, sofern sie uns jetzt bekannt sind, werden im Kapitel Ausblick (siehe Kapitel 10) erläutert. Es sollte bei diesem Versuch jedoch beachtet werden, dass SUIT dadurch sein primäres Ziel, früh in der Softwareentwicklung eingesetzt werden zu können, nicht wieder verliert.

9.4 Fazit

SUIT erfüllt in ersten Tests unsere Erwartungen und schient wenigstens für kleine Einplatzsysteme praxistauglich einsetzbar zu sein. Verteilte Benutzungssysteme sind mit SUIT noch nicht modellierbar. Der Code für den Aufruf der Folgemaske wird mit SUIT noch nicht automatisch geniert, da der Ort für die Erstellung nur vom Programmierer selbst sinnvoll bestimmt werden kann.

Durch die einfache SUIT Zustandsdiagramm-Notation ermöglich SUIT dem Anwender Benutzungsschnittstellen möglichst in den frühen Etappen der Softwareentwicklung einzusetzen, ohne sein Zielsystem und dessen Anforderungen vollständig kennen zu müssen. Diese überschaubare Notation ermöglich es auch, dem Anwender die Methodik Benutzungsschnittstellen in Form von Zustandsdiagrammen und Fensterentwürfen, ohne große Einarbeitung, umsetzen und verstehen zu können.

Durch das nicht Berücksichtigen der Funktionalität in der Modellierung kann es dazu kommen, dass in späteren Etappen der Softwareentwicklung, in denen die Funktionen ausimplementiert werden, die Konsistenz zwischen den Zustandsdiagramm-Modell und der Benutzungsschnittstelle verloren geht. Diese Konsistenzerhaltung und die dafür nötigen Erweiterungen in der SUIT-Zustandsdiagramm-Notation sollen im Kapitel Ausblick erläutert werden.

Wie SUIT sich letztendlich bei großen Projekten bewährt, wird sich erst dann zeigen, wenn SUIT wirklich zum praktischen Einsatz in großen Projekten kommt. Doch die Ergebnisse der Usability Tests und der Fallstudie lassen Positives erhoffen.

10. Ausblick

10.1 Einleitung

In diesem Kapitel soll ein Ausblick darüber gegeben werden, wie sich SUIT weiterentwickeln lässt. SUIT erfüllt in der aktuellen Version die von uns erhofften Ziele, die wir in der Vision (siehe 1.2) beschrieben haben. Benutzungsschnittstellen lassen sich in SUIT mit Hilfe von Zustandsdiagrammen und Fensterentwürfen modellieren, und das schon recht in den frühen Etappen der Softwareentwicklung. Die Simulationskomponente ermöglich es den Kunden einen Prototyp schnell vorzulegen und die Fassaden-Generierung nimmt dem Entwickler Programmierarbeit ab und bietet eine gute Schnittstelle zur Weiterentwicklung der mit SUIT modellierten Prototypen.

Für die nächste SUIT Version sollen die Aspekte der Konsistenzhaltung und Hierarchie neue primäre Ziele sein. Um diese Ziele zu erreichen, bedarf es Erweiterungen an SUIT, die in Abschnitt 10.2 erläutert werden. Im Abschnitt 10.3 folgt dann ein Fazit.

10.2 Erweiterungen von SUIT

Für die folgenden SUIT Versionen können zwei primäre Ziele gesetzt werden. Zunächst soll die Möglichkeit geboten werden, das gesamte Zustandsdiagramm Modell aufzuteilen und somit besser zu strukturieren. Diese Erweiterung soll in Abschnitt 10.2.1 erläutert werden. Abschnitt 10.2.2 bezieht sich dann auf Erweiterungen an SUIT, die auf die Konsistenzerhaltung zwischen dem SUIT-Zustandsdiagramm-Modell der tatsächlichen Benutzungsschnittstelle zielen.

10.2.1 Erweiterung Hierarchie

SUIT wurde bisher nur bei kleineren Projekten getestet. In diesen Projekten hat ein Zustandsdiagramm gereicht, um die gesamte Benutzungsschnittstelle zu beschreiben, ohne dass es unübersichtlich wurde. Bei größeren Benutzungsschnittstellen kann diese Übersichtlichkeit im Zustandsdiagramm-Modell verloren gehen. Um Übersichtlichkeit zu gewährleisten, muss eine Möglichkeit zur logischen Strukturierung des SUIT-Zustandsdiagramm-Modell geboten werden. Diese Strukturierung kann in Form von Hierarchien realisiert werden. Hierarchien bieten hierbei eine Möglichkeit, ein Zustandsdiagramm in mehrere Teildiagramme zu unterteilen.

Das aktuelle SUIT Zustandsdiagramm hat den Aspekt der Hierarchie noch nicht integriert. Da SUIT sich aber an Statechart-Definition anlehnt, kann dies eingebaut werden. Statechart bietet die Möglichkeit Hierarchien [Harel 1987] zu definieren und zu erzeugen. In der nächsten SUIT Version sollte – um Struktur und Übersichtlichkeit einzuführen – diese Hierarchie in die SUIT-Zustandsdiagramm-Notation eingeführt werden.

10.2.2 Erweiterung Konsistenzerhaltung

Die aktuelle SUIT Version berücksichtigt nicht die Implementierungen von Funktionalen Aspekten der Benutzungsschnittstelle. Das primäre Ziel von SUIT ist das frühe Prototyping. In dieser frühen Phase hat man noch kein detailliertes Verständnis über die Funktionalität. Daher wurden in der aktuellen SUIT-Zustandsdiagramm-Notation keinerlei Konstrukte angeboten, die diese Funktionalitäten im Modell berücksichtigen. In den späteren Entwicklungsphasen kommt man jedoch nicht an der funktionalen Implementierung vorbei. Werden vom SUIT-Anwender in diesen Phasen Funktionen ausimplementiert, die auf den Dialogverlauf der Benutzungsschnittstelle Einfluss nehmen, so sind diese nicht in dem Modell abbildbar.

In diesem Abschnitt widmen wir uns möglichen Konsistenzverlust-Ursachen und Notationserweiterung. So kann ein neues primäres Ziel der nächsten SUIT Version darin liegen, Konsistenzerhaltung für die späteren Etappen der Softwareentwicklung zu gewährleisten, um die Weiterentwicklung des mit SUIT erzeugten Prototyps zu optimieren. In Abschnitt 10.2.2.1 wird auf die Konsistenzverlust-Ursache eingegangen, die durch dialogablaufsteuernde Aktionsrückgabewerte verursacht werden. Abschnitt 10.2.2.2 geht dann auf die Konsistenzverlust-Ursache Inkarnationen ein.

10.2.2.1 Dialogsverlaufsteuernde Aktionsrückgabewerte

Ein Hauptgrund für diesen Konsistenzverlust in der Implementierungsphase sehen wir in der fehlenden Möglichkeit, dialogsverlaufsteuernde Aktionsrückgabewerte zu modellieren. Rückgabewerte von Aktionen mögen zum Zeitpunkt der Analyse noch nicht zu hundert Prozent bekannt sein, sie können jedoch Einfluss auf den Dialogverlauf haben. Eine totale Missachtung der funktionalen Implementierung vom Modell führt daher leicht zu einem Konsistenzverlust des Models. Um das Problem der dialogsverlaufsteuernden Rückgabewerte einer Aktion zu veranschaulichen, soll das Beispiel der benutzerrechteabhängigen Benutzungsschnittstelle dienen.

Bei einigen Benutzungsschnittstellen ist der Bildschirmmaskenverlauf abhängig davon, welcher Benutzer gerade am Arbeitsplatz arbeitet. Abhängig von seinen Benutzerrechten bekommt der Benutzer andere Bildschirmmasken angezeigt. Diese Benutzungsschnittstellensysteme bieten meist einen Anmeldebildschirm an, um den Benutzer zu verifizieren. Je nach Zugriffsrecht bekommt der angemeldete Benutzer dann unterschiedliche Masken zu sehen. Ist die Verifizierung des Benutzers nicht erfolgreich, wird in den meisten benutzerrechteahängigen Benutzungsschnittstellen der Zugriff auf die Masken verweigert, und der Benutzer bleibt beim Anmeldebildschirm. So kann dieser Dialogverlauf mit Interaktionsdiagrammen wie in Abb. 56 dargestellt werden.

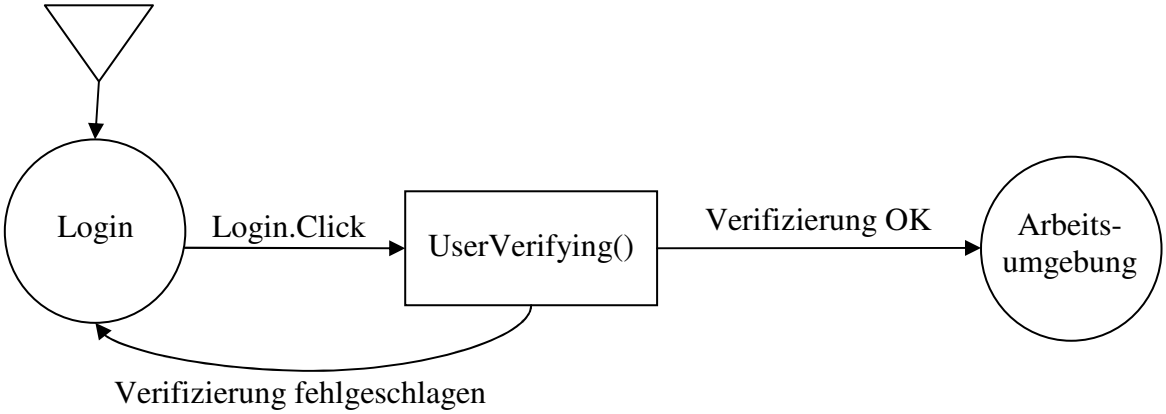


Abb. 56 Darstellung eines Login-Verfahren in Form eines Interaktionsdiagrammes

Diese benutzerrechteabhängigen Benutzungsschnittstellen lassen sich jedoch nicht mit der aktuellen SUIT-Zustandsdiagramm Notation darstellen. Der Dialogablauf wird also vom Rückgabewert der Verifizierungsfunktion, in Abb. 56 ***UserVerifying***(), gesteuert. Ein Zustandsdiagramm, das so einen Ablauf beschreiben könnte, müsste also um die Informationen des Rückgabewertes der Verifizierungsfunktion erweitert werden.

In der aktuellen SUIT Version ließe sich dieser Dialogverlauf nur wie in Abb. 57 darstellen.

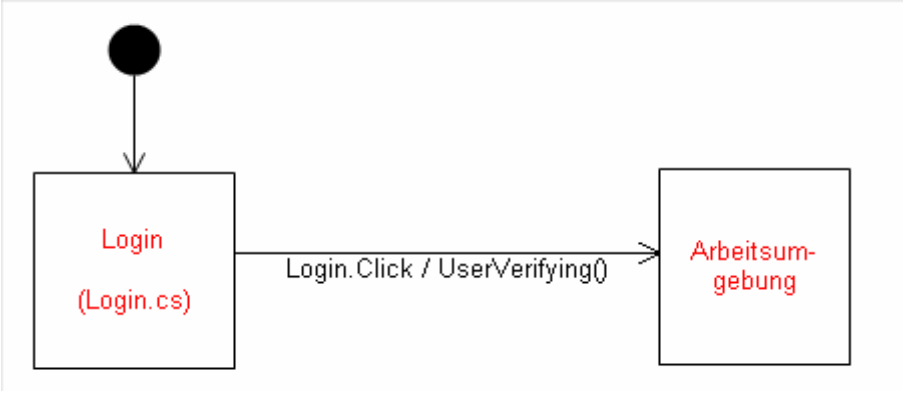


Abb. 57 Darstellung eines Login-Verfahrens in SUIT-Zustandsdiagramm Notation

Abb. 57 zeigt, dass mit dem Ereignis Login.Click die Aktion ***UserVerifying***() ausgeführt wird. Jedoch ist hier unabhängig vom Rückgabewert der Funktion ***UserVerifying***() der Folgezustand immer die Arbeitsumgebung. Somit ist – egal ob nun die Verifizierung OK oder nicht OK war - der Anmeldevorgang abgeschlossen. Dies ist für ein frühes Prototyping, wo es nur darum geht, den Kunden eine funktionalitätsfreie Benutzungsschnittstelle vorzulegen, noch akzeptabel, aber zum Beschreiben des endgültigen Dialogablauf des Benutzungsschnittstellensystem reicht dieses Diagramm nicht mehr aus.

Da SUIT das Ziel verfolgt, möglichst in den frühen Phasen der Softwareentwicklung zum Einsatz zu kommen und unter anderem möglichst früh den Kunden einen Prototyp dank der Simulationskomponente vorlegen zu können, ist die SUIT-Zustandsdiagramm Notation ausreichend. Die Gefahr bei dieser Notation ist jedoch, dass in den späteren Phasen es zu einem Konsistenzverlust zwischen Diagramm und Benutzungsschnittstellen kommt. Will man den Dialogablauf des Login-Vorgang dennoch in SUIT darstellen, könnte ein Diagramm wie in Abb. 58 entstehen, das mit Vorsicht zu genießen ist.

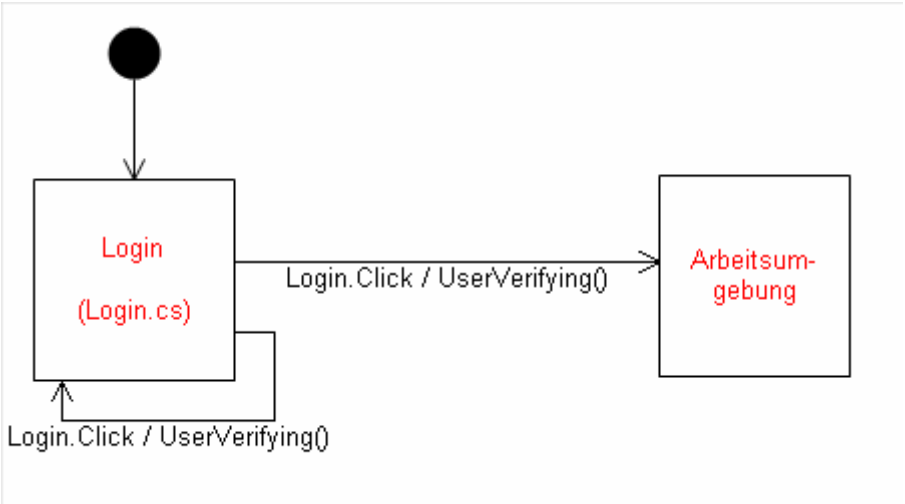


Abb. 58 Login-Verfahren in SUIT-Zustandsdiagramm Notation (Erweitert)

Abbildung 3 zeigt nun, dass das Ereignis Login.Click die Aktion ***UserVerifying***() startet, und diese Aktion sowohl zum Folgezustand Arbeitsumgebung, als auch rekursiv zu sich selbst führt. Durch die zwei Transitionen, die beide auf das Ereignis Login.Click reagieren und beide die Aktion ***UserVerifying***() anstoßen, jedoch andere Folgezustände aufweisen, wird hier der verzweigte Dialogablauf dargestellt. Allerdings darf man sich hier nicht irreführen lassen, denn dieses Diagramm beschreibt nicht den Sachverhalt, der in Abb. 56 durch ein Interaktionsdiagramm beschrieben worden ist. Auch in dem Diagramm in Abb. 58 ist keinerlei Semantik über den Rückgabewert der ***UserVerifying***() Funktion enthalten und würde im Falle einer Simulation sogar dazu führen, dass völlig unklar wäre, welche der Zustände – und somit welche Maske – nun im Falle eines Login.Click Ereignis aktiv wäre. Daher sollte der SUIT-Anwender möglichst solche Diagramme vermeiden. Der SUIT - Zustandsdiagrammeditor erlaubt zwar die Erstellung solcher Transitionen, die auf dasselbe Ereignis reagieren, aber zu unterschiedlichen Folgezustände führen. Der SUIT-Anwender sollte jedoch hiervon absehen, da es sonst zu unvorhersehbaren Problemen in der Simulation kommen kann.

Der SUIT-Zustandsdiagrammeditor in der aktuellen Version hat keinerlei Sicherheitsabfragen gegen solche unklaren Dialogverzweigungen. Diese Sicherheitsabfrage wäre eine mögliche Anforderung der nächsten SUIT Version.

Zusammenfassend ist mit der aktuellen SUIT-Zustandsdiagramm-Notation der Dialogverlauf eines Login-Verfahren einer Benutzungsschnittstelle nicht komplett darstellbar. Für einen frühen Prototyp reicht aber die Darstellungsform, indem man von dem Rückgabewert der Verifizierungsfunktion absieht und den Anmeldevorgang als immer erfolgreich ansieht.

Der Rückgabewert einer Aktion ist für die Modellierung des Zustandsdiagramm in SUIT ohne Belang. Durch diese fehlende Möglichkeit kann das Modell in den späteren Phasen nicht mehr exakt die Benutzungsschnittstelle beschreiben, wenn diese dialogablaufsteuernde Aktionsrückgabewerte vom Programmierer selbst einprogrammiert werden müssen. Eine Erweiterung der aktuellen SUIT-Zustandsdiagramm-Notation müsste also in Betracht gezogen werden, wenn für die nächste SUIT-Version die Konsistenzerhaltung primäres Ziel sein soll.

Eine mögliche Erweiterung der Notation um dialogablaufsteuernde Aktionsrückgabewerte im Modell modellieren zu können, wäre die Wächter-Bedingung, die in [Harel 1987] für

Statecharts definiert sind. Eine Wächterbedingung ist eine Bedingung die wahr sein muss, damit der Übergang, zu dem sie gehört, stattfinden kann. Mit Wächterbedingungen kann dokumentiert werden, dass ein bestimmtes Ereignis je nach Bedingung zu unterschiedlichen Übergängen führt. Für das Login-Verfahren-Beispiel könnte eine Diagramm mit Wächterbedingung wie in Abb. 59 aussehen:

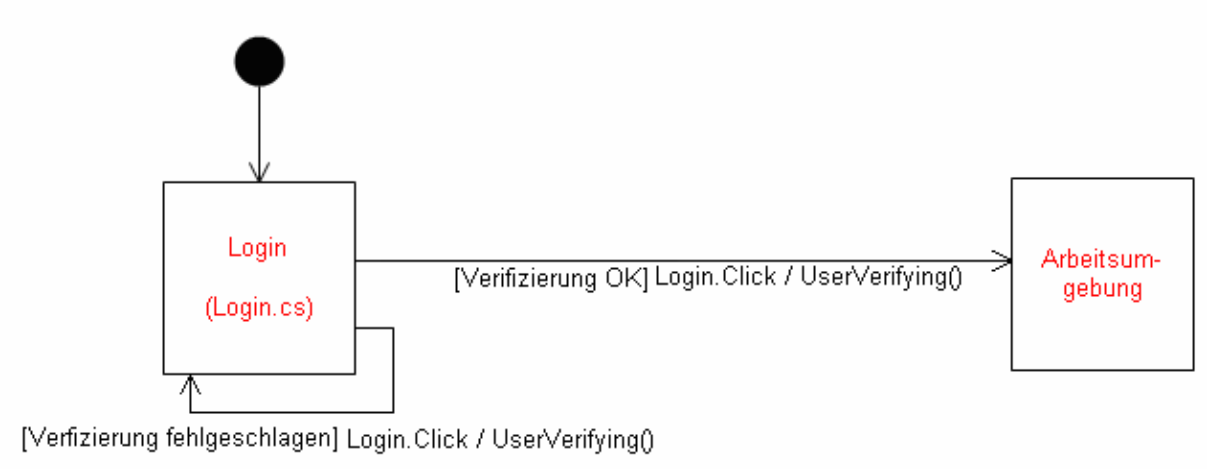


Abb. 59 Login-Verfahren mit Wächterbedingung

In Falle der dialogablaufsteuernden Rückgabewerte, wäre die Wächterbedingung also das Erfüllen eines bestimmten Rückgabewertes. Bei dieser Notationserweiterung sollte jedoch bedacht werden, dass diese Erweiterung auch in der Simulation berücksichtig werden sollte. So muss in der Simulation diese Rückgabewerte irgendwie schon zurückgeliefert werden. Dies setzt teilweise eine Implementierung der Aktionen voraus. Inwiefern diese Implementierung den frühen Einsatz der Simulation und somit das Prototyping in der frühen Etappen der Softwareentwicklung behindert, muss noch in Rahmen einer ausführlichen Analyse untersucht werden. Optimal wäre, wenn diese Wächterfunktionen als optional angesehen werden, so dass diese funktionalen Details zur den frühen Etappen der Softwareentwicklung noch nicht modelliert werden muss.

10.2.2.2 Inkarnationen der Masken

In Kapitel 3.2.5 wurde der Dialogwechsel nach Denert beschrieben. Dort wurde der Begriff Inkarnationen von Masken eingeführt und erläutert. Diese Möglichkeit, die Inkarnationen der Masken zu definieren, ist in der aktuellen SUIT-Zustandsdiagrammnotation noch nicht vorgesehen.

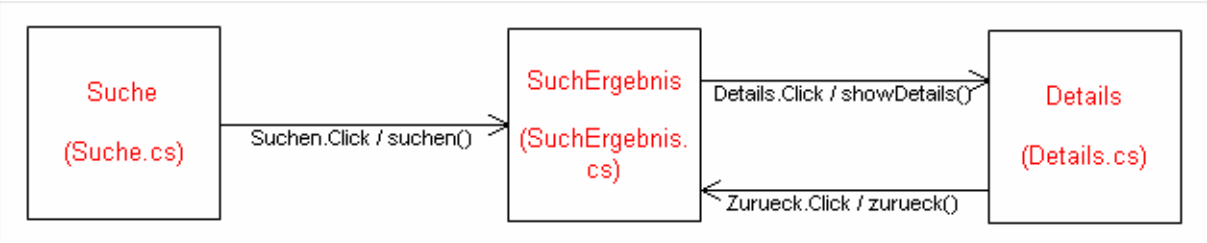


Abb. 60 Inkarnationsbeispiel Zustandsdiagramm

Abb. 60 beschreibt eine allgemeine Suche. In Dialog Suche wird nach Eingabe eines Suchwortes und Betätigen des Suchbuttons in die Folgemaske SuchErgebnis gewechselt. In dieser Maske werden alle passenden Suchergebnisse aufgelistet. Nach Auswahl eines

bestimmten Suchergebnisses und Betätigen der Taste Details wird zur Detailansicht gewechselt. Beim Zurückkehren in dem Dialog der Suchergebnisse ist es wünschenswert, wenn dieser Dialog noch offen ist, so dass eine Rückkehr zu den in ihm bereits angezeigten Suchergebnissen möglich ist. Die Daten der Suche dürfen also nicht verloren gehen. Es muss der jeweils aktuelle Stand zwischengespeichert werden. Kehrt der Benutzer zu ihm zurück, befindet sich der Dialog weiterhin in genau diesem gespeicherten Zustand. Um dies erreichen zu können muss für den Dialog der Suchergebnisse eine Inkarnation definiert werden.

Mit der Definition von Inkarnationen wird vorgegeben, ob beim Unterbrechen eines Dialogs der jeweils aktuelle Zustand zwischengespeichert wird oder nicht. Hier in unserem Beispiel z.B. ist es sinnvoll beim Zurückkehren von Details zu SuchErgebnis die Suchergebnisse noch anzeigbar sind. Im Dialog Details können hingegen die Daten, also die Anzeige der Details eines Suchergebnisses verloren gehen, also nicht zwischengespeichert werden. Es muss also die Möglichkeit gegeben werden, abhängig vom Dialog eine Inkarnation festzulegen.

Für eine frühe Simulation und damit ein frühes Prototyping ist die Betrachtungsweise der Inkarnationen der Masken noch unnötig. Teilweise ist es in dieser frühen Phase sogar noch nicht mal bekannt, ob und welche Inkarnationen für eine Maske zu erwarten sind. In den späteren Etappen der Softwareentwicklung ist es jedoch in einigen Fällen unerlässlich, Inkarnationen für die Masken zu bestimmen, sofern aus dem Zustandsdiagrammmodell der Code für Dialogwechsel automatisch generiert bzw. zur Laufzeit der Dialogwechsel bestimmt werden soll.

Da die aktuelle SUIT Version den Dialogwechseln nicht automatisiert und diese Aufgabe dem SUIT-Anwender überlässt (siehe dazu Abschnitt 9.2.4), ist die aktuelle Zustandsdiagrammnotation ohne die Informationen über die Inkarnationen der Masken völlig ausreichend.

Die fehlende Möglichkeit mit der SUIT-Zustandsdiagrammnotation Informationen über die Inkarnationen der Masken abzulegen, kann, sobald der SUIT-Anwender die Dialogwechsel in seinem Benutzungsschnittstellensystem einprogrammiert, führen dazu, dass das Zustandsdiagrammmodel seine Konsistenz verliert. In den späteren Phasen ist also damit zu rechnen, dass die Konsistenz zwischen Modell und Benutzungsschnittstelle dadurch verloren geht.

10.3 Fazit

Die nächste Version von SUIT kann dahingehen erweitert werden, das sie Hierarchien unterstützt. Dies bietet dem Anwender die Möglichkeit sein Modell zu strukturieren um Übersichtlichkeit in größeren Projekten zu gewährleisten. Die Konsistenzerhaltung zwischen dem Modell und der Benutzungsschnittstellen könnte ein weiteres wichtiges Ziel der nächsten SUIT Version sein. Hierfür müsste die aktuelle SUIT-Zustandsdiagramm-Notation um die Modellierungsmöglichkeit von dialogablaufsteuernden Aktionsrückgabewerte und Inkarnationen der Masken erweitert werden. Dialogablaufsteuernde Aktionsrückgabewerte können mit der Einführung der Statecharts Wächter-Bedingung [Harel 1987] realisiert werden. Es sollte bei diesen Erweiterung immer berücksichtig werden, dass die aktuellen positiven Eigenschaften von SUIT, wie z.B. die Möglichkeit in den frühen Etappen der Softwareentwicklung zu Simulieren, nicht verloren gehen.

11. Resumé

Im Rahmen dieser Bachelor-Arbeit wurde SUIT entwickelt. In SUIT kann das dynamische Verhalten einer Benutzungsschnittstelle mit Hilfe von Zustandsdiagrammen und Fensterentwürfen modelliert werden. Das Modell einer Benutzungsschnittstelle kann in SUIT simuliert und aus dem Modell automatisch Code in Form einer Fassade generiert werden. Die Modellierung von Benutzungsschnittstellen mit endlichen Automaten und Fensterentwürfen, wird in der Literatur – abgesehen von ein paar Ausnahmen wie Denert in [Denert 1992]- kaum angesprochen. Auch ergab die im Rahmen der Marktanalyse durchgeführte Suche nach einem existierenden Tool, dass die Methode der Benutzerschnittstellenmodellierung umsetzt, kein Ergebnis. Es zeigte sich dadurch die Notwendigkeit, SUIT zu realisieren.

Der Einsatz von SUIT in einer Fallstudie am Beispiel eines Pizzalieferservice-Systems hat sich als praxistauglich erwiesen. Neben einer Fallstudie wurde SUIT noch in einem Usability-Labor von freiwilligen Testern getestet. Die Tester äußerten sich positiv über die Vorgehensweise, Benutzungsschnittstellen mit Hilfe von Zustandsdiagrammen und Fensterentwürfen zu modellieren. Alle Testpersonen sind der Meinung, dass diese Methode eine große Hilfe für die Erstellung von Benutzungsschnittstellenprototypen darstellt und ihnen manuelle Arbeit in größerem Umfang erspart.

Die in der Vision geschilderten Erwartungen an SUIT haben sich im Usability-Labor bestätigt. Die Tester können sich gut vorstellen, dass sich durch die schnelle Prototyperzeugung in SUIT die Benutzer-Entwickler-Kommunikation verbessern würde. Auch die Fassadengenerierung, die eine Schnittstelle zwischen der Interaktionskomponente und der funktionalen Komponente darstellt, wird von den Testern als positiv eingestuft. Die automatische Fassadengenerierung bietet somit eine Schnittstelle zur Weiterentwicklung des Prototyps und nimmt den Programmierern Programmieraufwand ab.

Die in Rahmen dieser Bachelor-Arbeit durchgeführten Tests an SUIT deuten darauf hin, dass SUIT die gewünschten Aussichten, die in der Vision beschrieben wurden, erfüllt. Wie sich SUIT letztendlich bei großen Benutzungsschnittstellen bewährt, lässt sich erst sagen, wenn SUIT in der Praxis wirklich zum Einsatz kommt. Die ersten Tests lassen jedoch Positives erhoffen.

Quellen

[Denert 1992] Ernst Denert, Software-Engineering, Springer Verlag, 1992

[eclipse] http://www.eclipse.org, 11.11.2006

[Gamma 2004] Erich Gamma, Richard Helm, Ralph E. Johnson: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, Addison Wesley in Pearson Education Deutschland, 2004

[Harel 1987]David Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 1987.

[Jigloo3.9.5] http://www.cloudgarden.com/jigloo, v.3.9.5 11.11.2006

[Together] http://www.borland.com/us/products/together/index.html, 11.11.2006

[MyersRosson92] B.A. Myers and M.B. Rosson; Survey on user interface programming. In SIGCHI'92: Human Factors in Computing Systems, May 1992.

[Visio2003] http://office.microsoft.com/de-de/visio/default.aspx, 11.11.2006

[Visual Studio.Net 2003] http://www.microsoft.com/germany/msdn/library/visualtools/UeberblickUeberVisualStudioNET2003.mspx?mfr=true, 11.11.2006

[Züllighoven 1998] Züllighoven, Heinz, Das objektorientierte Konstruktionshandbuch, Heidelberg : dpunkt-Verl., 1998

[CodeDom] http://windowsforms.net/articles/shapedesigner.aspx, 12.01.07

[uselab] http://users.informatik.haw-hamburg.de/~use-lab, 12.01.07

[msdn-attribute] http://msdn2.microsoft.com/de-de/library/system.attribute(VS.80).aspx, 12.01.07

[WindowBuilderPro] http://www.windowbuilderpro.com, 13.01.07

[VisualEditor] http://www.eclipse.org/vep/WebContent/main.php, 13.01.07

Abbildungsverzeichnis

Abb. 1 Literatur-Suche 8
Abb. 2 Zustandsdiagramm Literatur Suche 9
Abb. 3 Zustandsdiagramm Literatur Suche um die Ereignisse angereichert 10
Abb. 4 Zustandsdiagramm Literatur Suche um die Maskenskizzen angereichert 10
Abb. 5 Zustandsdiagramm Literatur Suche um die Masken angereichert 11
Abb. 6 Zustandsdiagramm Literatur Suche um die Aktionen angereichert 12
Abb. 7 Fassade 13
Abb. 8 Einsatz von SUIT 15
Abb. 9 Jigloo in Eclipse 19
Abb. 10 Die Palette in Jigloo 20
Abb. 11 Property Fenster in Jigloo 20
Abb. 12 Von Jigloo generierter Code-Ausschnitt einer GUI mit 2 JButtons 21
Abb. 13 Code-Ausschnitt einer GUI mit 2 JButtons und ActionListener in Jigloo 22
Abb. 14 integrierter GUI-Builder von Microsoft Visual Studio.Net 2003 23
Abb. 15 Together 25
Abb. 16 Visio 26
Abb. 17 Interaktionsdiagramm eines Geldausgabeautomaten aus [Denert 1992] 32
Abb. 18 Spezifikation einer Aktion mittels Entscheidungstabelle [Denert 1992] 36
Abb. 19 Zustand 39
Abb. 20 Startzustand 39
Abb. 21 Zustandsübergang 40
Abb. 22 Rekursiver Zustandsübergang 40
Abb. 23 Integration von SUIT in die Microsoft Visual Studio.Net Architektur 49
Abb. 24 Zusammenspiel der SUIT Komponenten 50
Abb. 25 Vererbungsbaum der Elemente in SUIT Zustandsdiagramm-Designer 51
Abb. 26 Architektur des Zielsystem mit Fassade 53
Abb. 27 Ausschnitt aus dem Klassendiagramm 57
Abb. 28 Toolbox vor dem Hinzufügen der Toolboxitems für das Zustandsdiagramm 60
Abb. 29 Toolbox nach dem Hinzufügen der Toolboxitems für das Zustandsdiagramm 60
Abb. 30 Zustandsdiagramm 63
Abb. 31 Eigenschaftsfenster im Zustandsdiagrammeditor 64
Abb. 32 Visual Studio.net in der Ansicht von SUIT 67
Abb. 33 SUIT-Zustandsdiagramm mit DragFeedback 72
Abb. 34 Dragfeedback 73
Abb. 35 Anwendungsfalldiagramm Pizzaservice 79
Abb. 36 Diagramm der Benutzungsschnittstelle des Angebotsverwaltungswerkzeugs 80
Abb. 37 Maske der Angebotsuche 81
Abb. 38 Maske der Angebotanzeige 82
Abb. 39 Maske der Angebotserstellung 82
Abb. 40 Maske der Angebotsdatenpflege 82
Abb. 41 Diagramm der Benutzungsschnittstelle des Kundenverwaltungswerkzeugs 83
Abb. 42 Maske der Kundensuche 84
Abb. 43 Maske der Kundenanzeigen 84
Abb. 44 Maske der Kundenerstellung 85
Abb. 45 Maske der Kundenpflege 85

Abb. 46 Maske der Bestellung 86
Abb. 47 Startmaske des Gesamtsystems Pizzaservice 87
Abb. 48 Diagramm der Benutzungsschnittstelle des Gesamtsystems 88
Abb. 49 Ein Ausschnitt der Fassade zum Zustandsdiagramm in Abb. 48 89
Abb. 50 Pizzasystem in der Simulationsansicht von SUIT 89
Abb. 51 Lösung der ersten Testperson 94
Abb. 52 Lösung der zweiten Testperson 95
Abb. 53 Lösung der dritten Testperson 96
Abb. 54 Lösung der vierten Testperson 97
Abb. 55 SUIT-Zustandsdiagrammbeispiel 101
Abb. 56 Darstellung eines Login-Verfahren in Form eines Interaktionsdiagrammes 108
Abb. 57 Darstellung eines Login-Verfahren in SUIT-Zustandsdiagramm Notation 108
Abb. 58 Login-Verfahren in SUIT-Zustandsdiagramm Notation (Erweitert) 109

Tabellenverzeichnis

Tabelle 1 Notwendige Anforderungen an SUIT 18
Tabelle 2 Neu erkannte Anforderungen aus Visio und Together 28
Tabelle 3 Die Anforderungen an SUIT 46
Tabelle 4 Parameter von draw 66
Tabelle 5 Parameter von DrawCreateDragFeedback 68
Tabelle 6 Parameter von delete 69
Tabelle 7 Parameter von initialize 69
Tabelle 8 Statische Variablen der Klasse ZDiagrammElementDesigner 70
Tabelle 9 Parameter von GetHitTest 71
Tabelle 10 Parameter von IsInRect 71
Tabelle 11 Parameter von GetCursor 72
Tabelle 12 Parameter von DrawAdornments 73
Tabelle 13 Parameter von DrawDragFeedback 74
Tabelle 14 Parameter von Drag 74
Tabelle 15 Parameter von onClick 75

Index

A

Aktion 12, 13, 30, 31, 33, 34, 35, 36, 37, 38, 40
Aktivität 11, 12, 38, 39
Anwendungsfalldiagramm 76, 78, 79
Architektur 48, 49, 50, 51, 52, 53

B

Benutzereingaben 8, 13, 36, 40
Benutzungsschnittstelle 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 28, 30, 31, 34, 37, 38, 40, 41, 76, 78, 80, 83, 84, 86, 88, 90, 100, 101, 104, 105, 106, 107, 108, 111, 112
Bildschirmmasken 12, 14, 16, 38

C

CodeDom 50, 51, 52, 61, 62, 63
Codegenerierung 12, 14, 21, 24, 50, 51, 52, 61, 90, 93, 95, 101, 102, 103

D

Dialogwechsel 34

E

Eclipse 18, 19, 24
Elementar Werkzeug 15, 78, 79, 80, 87
Endlicher Automat 9, 30, 31, 40
Endzustand 38, 39
Entscheidungstabelle 33, 37, 40
EnvDTE 48, 63, 65, 66

F

Fallstudie 76
Fassade 13, 14, 17, 18, 31, 37, 40, 41, 46, 52, 53, 54, 58, 63, 64, 88, 89, 90, 91, 92, 95, 98, 99, 106, 112
Fassaden-Pattern 52
Fensterentwurf 13, 14, 15, 16, 17, 18, 28, 30, 31, 37, 40, 76, 80, 86, 90, 91, 92, 95, 97, 99, 105, 106, 112
FileCodeModel 63
Funktionale Komponente 13, 53

G

GUI-Builder 11, 12, 14, 15, 16, 17, 18, 19, 22, 23, 24, 25, 28, 29, 31, 41, 48, 49, 50, 51, 52, 59, 61, 65, 93, 100
GUI-Komponente 10, 20, 21, 22, 23, 24

I

Inkarnation ……..34, 35, 107, 110, 111

Interaktionsdiagramm ……31, 32, 33, 34, 37, 107, 108, 109
Interaktionskomponente……..…… 13, 31, 52, 53, 54, 99, 112

J

Jigloo ……19, 20, 21, 22

K

Klassendiagramm ……57

Kombi Werkzeug ……15, 78, 86, 87

Komponente ……4, 48, 49, 50, 51, 52, 53, 54
Konsistenzerhaltung ……104, 105, 106, 107, 109, 111

M

Maske …… 8, 9, 12, 14, 18, 21, 30, 34, 38, 39, 43, 44, 46, 51, 65, 80, 81, 82, 83, 84, 86, 101, 102, 103, 104, 109, 110, 111
Modellierung ……7, 9, 11, 14, 15, 17, 26, 30, 31, 34, 37, 40, 80, 92, 93, 94, 95, 97, 100, 104, 105, 109, 112

N

Navigation ……..30, 38
Notation ……15, 32, 37, 39

O

Operation ……11, 12, 17, 24, 38

P

Prototyping ……..……...14, 92, 99, 107, 108, 110, 111

S

Simulation ……14, 18, 31, 38, 41, 46, 49, 51, 52, 54, 58, 88, 89, 92, 93, 94, 95, 96, 99, 104, 106, 108, 109, 110, 111

Simulationsansicht …… 46, 56 89

Sprachunabhängigkeit ……..……61, 63

Startzustand ……..……31, 39, 58, 59, 61

Statechart ……..……13, 37, 46, 57, 106, 110, 111
SUIT ……13, 14, 15, 16, 17, 18, 28, 31, 37, 40, 41, 42, 47, 48, 49, 50, 51, 52, 53, 55, 57, 61, 75, 80, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112

T

Together ……25, 26, 27, 28

Toolbox ……51, 52, 55, 59, 60, 61, 94
Transition ……37, 38, 39, 40, 42, 55, 101, 102, 103 siehe Zustandsübergang

U

UML ……7, 37, 25, 57
Usability ……91, 97, 99, 105, 112

V

Virtuelle Taste..... 33, 34
Visio 26, 27, 28
Visual Editor 18
Visual Studio 18, 22,23, 24, 25, 28, 31, 41, 48, 49, 50, 52, 59, 60, 61, 65, 66, 93, 94, 95, 96, 97

W

Wächter 109, 110, 111
WAM 15, 77
Weiterentwicklung 7, 17, 51, 90, 106
Window Builder Pro 18

Z

Zustand ..... 11, 12, 13, 14, 17, 26, 30, 31, 33, 34, 35, 37, 38, 39, 45, ... 51, 58, 80, 83, 86, 90, 103, 104
Zustandsdiagramm 9, 10, 11, 12, 13, 14, 17, 18, 28, 37, 39, 41, 42, 51, 52, 53, 58, 61, 62, 63, 83, 86, 87, 88, 89, 90, 106, 108, 109
Zustandsdiagramm-Editor 17, 100
Zustandsübergang..... 12, 18, 37, 39, 40, 42, 51, 55, 56, 58
Zustandswechsel..... 9, 30

Anhang:

Benutzerhandbuch

SUIT Version 1.0

Stand: 06.01.2007

Inhaltsverzeichnis

1. Systemanforderung 123

2. Installationshinweise 124
2.1 Installation von SUIT 124
2.2 Installationen von Zustandsdiagramm-Template 124
2.2.1 Installation von C# Template 124

3. Einbindung von SUIT ins Projekt 125

4. Bedienung des SUIT - Zustandsdiagrammeditor 126
4.1 Erzeugung eines Zustandsdiagramm 126
4.2 Erzeugen eines Startzustand 127
4.3 Erzeugung eines Zustands 129
4.4 Erzeugung eines Zustandsübergang 130
4.5 GUI-Maske Zuweisung zu einem Zustand 131
4.6 GUI-Maske Zuweisung eines Zustands ändern 132
4.7 Zuweisung von Event und Action zu einem Zustandsübergang 133
4.8 Event und Action eines Zustandsübergang ändern 134
4.9 Löschen von Elementen aus dem Zustandsdiagramm 135

5. Fassadengenerierung 136

6. Simulation 137
6.1 Voraussetzung für die Simulation 137
6.2 Wie startet man eine Simulation 137

1. Systemanforderung

SUIT wurde für die Entwicklungsumgebung Visual Studio.Net 2003 von Microsoft entwickelt. Um SUIT zu benutzen, sind Microsoft Visual Studio.Net 2003 und das Microsoft .Net Framework Version 1.1 Voraussetzung. Getestet und entwickelt wurde SUIT mit dem .Net Framework 1.1.4322 SP1.

2. Installationshinweise

2.1 Installation von SUIT

Bitte beachten Sie vor der SUIT-Installation die Systemanforderungen in Abschnitt 1. Um SUIT zu Installieren, legen Sie die SUIT-CD in ihr CD-ROM Laufwerk ein und starten Sie die sich auf der CD befundenen Setup.exe. Folgen Sie den Installationsanweisungen der Setup-Routine, bis zum erfolgreichen Abschluss der SUIT-Installation.

2.2 Installationen des Zustandsdiagramm-Template

In der Aktuellen SUIT Version (Vers. 1.00) unterstützt SUIT zur Zeit nur für die .Net Sprache C# einen ProjectItem-Template zur Erzeugung von Zustandsdiagrammklassen. Daher gibt es auch nur eine Installaltionsbeschreibung für das Einrichten des Template für C# (siehe 2.2.1).

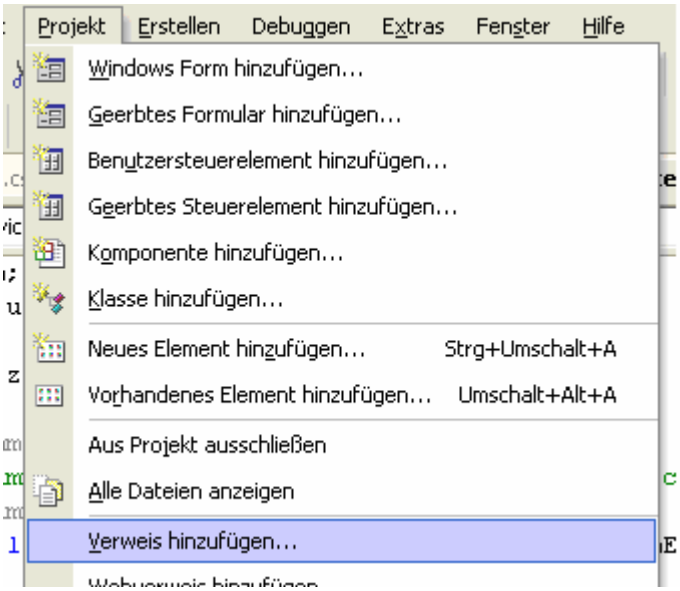
2.2.1 Installation von C# Template

Um für C# Projekte das ProjektItem BenutzerschnittstellenZustandsdiagramm zu bekommen, installieren Sie den C# Template. Kopieren Sie dazu das ganze Verzeichnis „C#“ auf der CD-ROM in ihren Visual Studio.Net 2003 Verzeichnis. Das Visual Studio.Net 2003 Verzeichnis finden Sie standardmäßig unter **C:\Programme\Microsoft Visual Studio .NET 2003**. Überschreiben Sie die vorhanden Dateien und Verzeichnisse, wenn der Kopiervorgang Sie darauf hinweist.

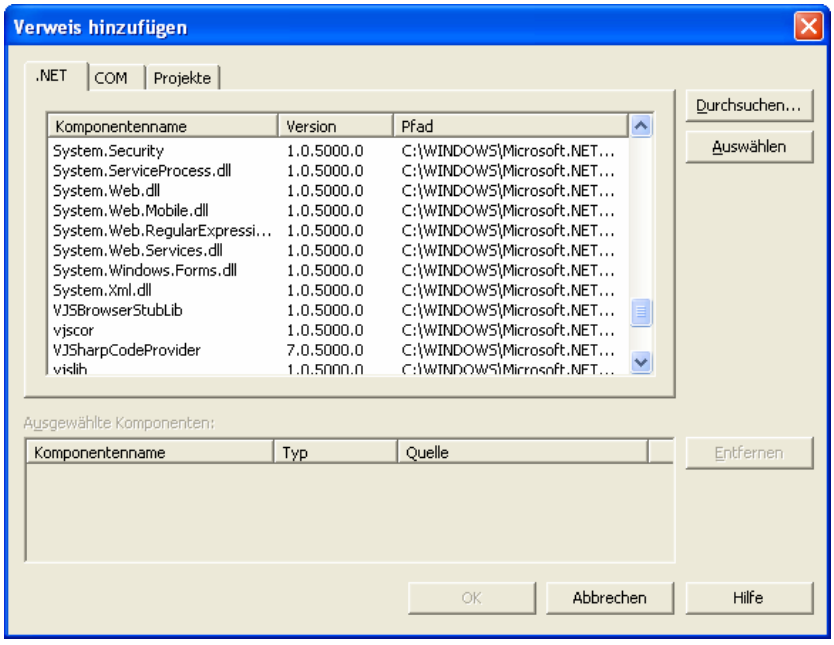
3. Einbindung von SUIT ins Projekt

1. Um SUIT in ihrem Projekt einzubinden, müssen Sie zuerst ihr Projekt laden oder neu anlegen.

2. Wählen Sie nun folgende Menüs aus:
   Projekt → Verweis hinzufügen…



3. Suchen Sie nun nach der die SUIT Library, indem Sie auf **„Durchsuchen…“** klicken (Standardmäßig unter: C:\Programme\SUIT\SUIT.dll). Wählen Sie SUIT.dll durch ein Doppelklick aus.



4. Bestätigen Sie Ihre Auswahl mit Klicken des OK-Buttons. Nun sollte SUIT in ihrem Projekt eingebunden sein.

4. Bedienung des SUIT–Zustandsdiagrammeditors

Der SUIT Zustandsdiagrammeditor in der aktuellen SUIT Version (Vers. 1.00) unterstützt alle .Net Sprachen. Zustandsdiagramm-Klassen können mit dem Editor in allen Sprachen erzeugt werden, jedoch geht das Generieren der Fassade und die Simulation in der vorliegende Version nur mit C# Projekten. Daher wird empfohlen diese SUIT Version einschließlich nur in C# Projekten zu verwenden.

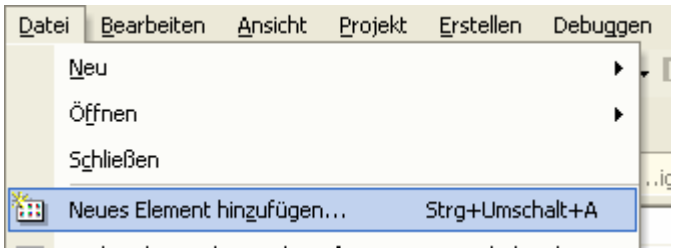
4.1 Erzeugung eines Zustandsdiagramm

Um ein SUIT Zustandsdiagramm zu erzeugen, erstellen Sie eine Klasse, die von **`SUIT.ZustandsdiagrammEditor.Zustandsdiagramm`** erbt. Um von **`SUIT.ZustandsdiagrammEditor.Zustandsdiagramm`** erben zu können, sollten Sie die SUIT Klassenbibliothek in ihrem Projekt eingebunden haben (siehe hierzu Abschnitt 3). Öffnen Sie diese Klasse in der Design-Ansicht mit Visual Studio.Net 2003, um den SUIT Zustandsdiagrammeditor aufzurufen. Sie haben nun die Möglichkeit, ihr Zustandsdiagramm zu entwerfen.

Wenn Sie – wie empfohlen – mit C# Projekten arbeiten, können Sie den Zustandsdiagramm-ProjektItem Template für C# Projekte installieren und ein Zustandsdiagramm per Template erzeugen lassen. Die Installation des C# Templates finden sie in Abschnitt 2.2.1 beschrieben.

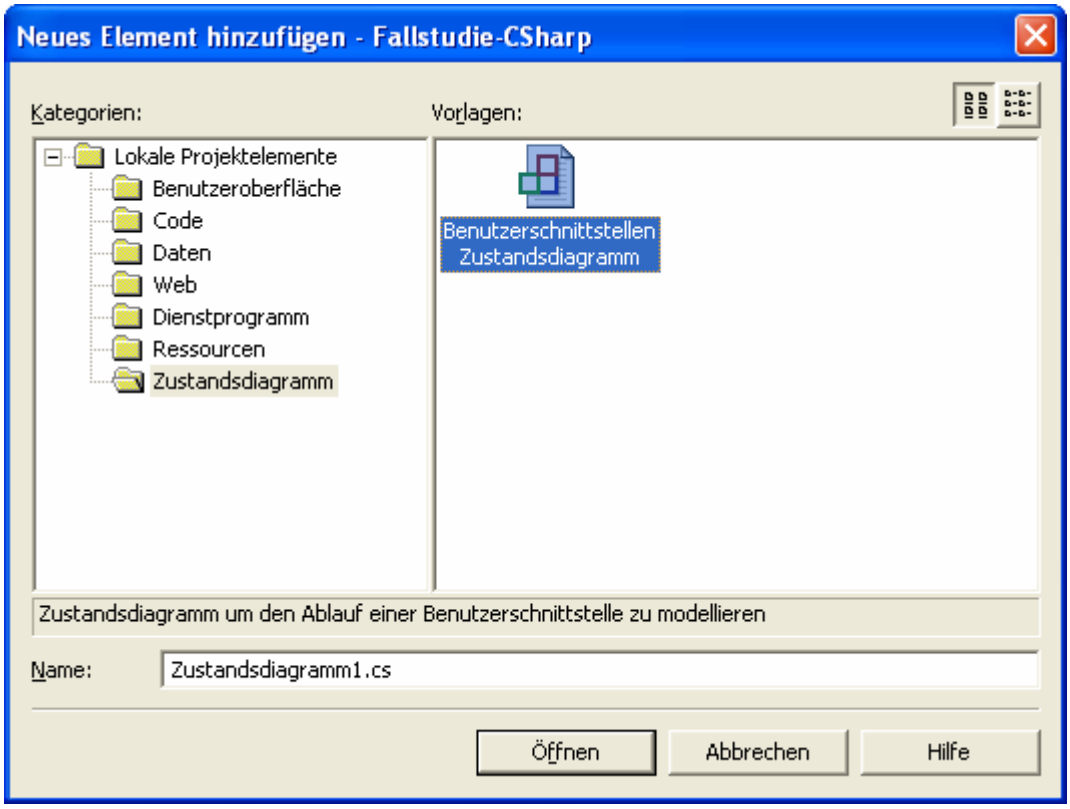
Um ein Zustandsdiagramm per Template zu erzeugen gehen Sie wie folgt vor:

1. Wählen Sie folgende Menü aus:
   Datei → Neues Element hinzufügen…



2. Es erscheint nun ein neues Dialog, in dem Sie eine Liste der Kategorien sehen. Wählen Sie aus dieser Liste folgenden Pfad:
   Lokale Projekte\Zustandsdiagramm

   Wählen Sie danach die Vorlage „Benutzerschnittstellen Zustandsdiagramm“ aus.



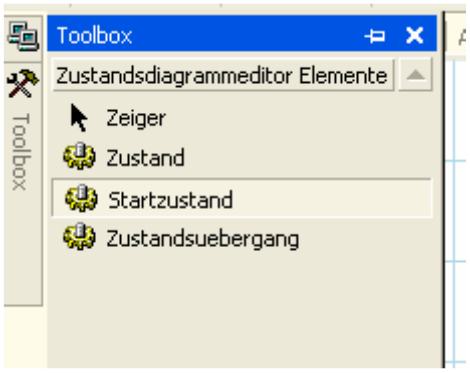
3. Geben Sie nun einen Namen für ihr Zustandsdiagramm ein, und bestätigen den Vorgang mit einem Klick auf den Button „Öffnen“. Sie sollten nun eine Zustandsdiagramm-Klasse in ihrem Projekt angelegt bekommen haben und den Designer-Ansicht für diese neue Klasse sehen.

4.2 Erzeugen eines Startzustand

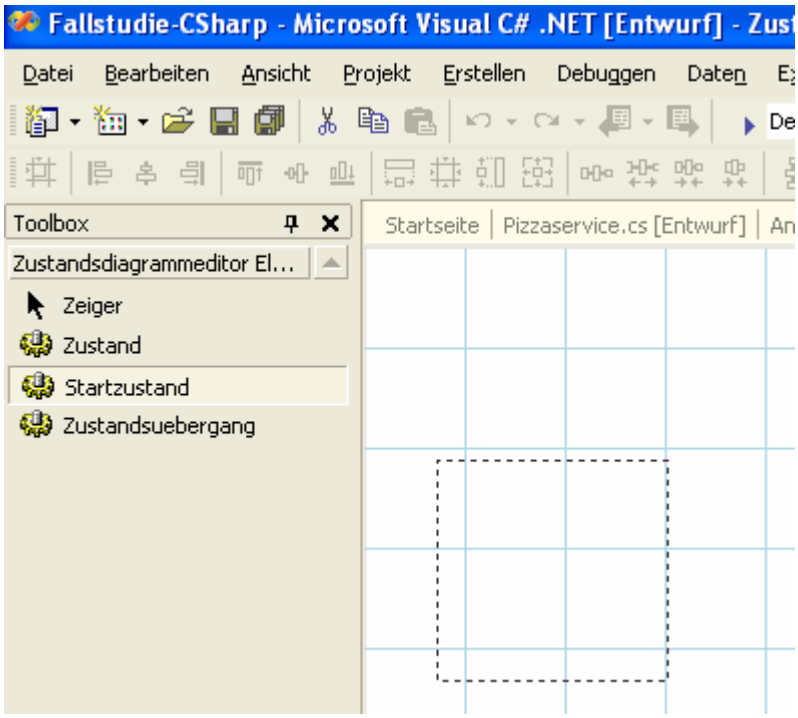
Bevor Sie einen Startzustand erzeugen können, müssen Sie ein Zustandsdiagramm erzeugt haben (Siehe 4.1). Öffnen Sie dieses Zustandsdiagramm in seiner Designer-Ansicht.

Um nun einen Startzustand per Drag&Drop zu erzeugen, gehen Sie wie folgt vor:

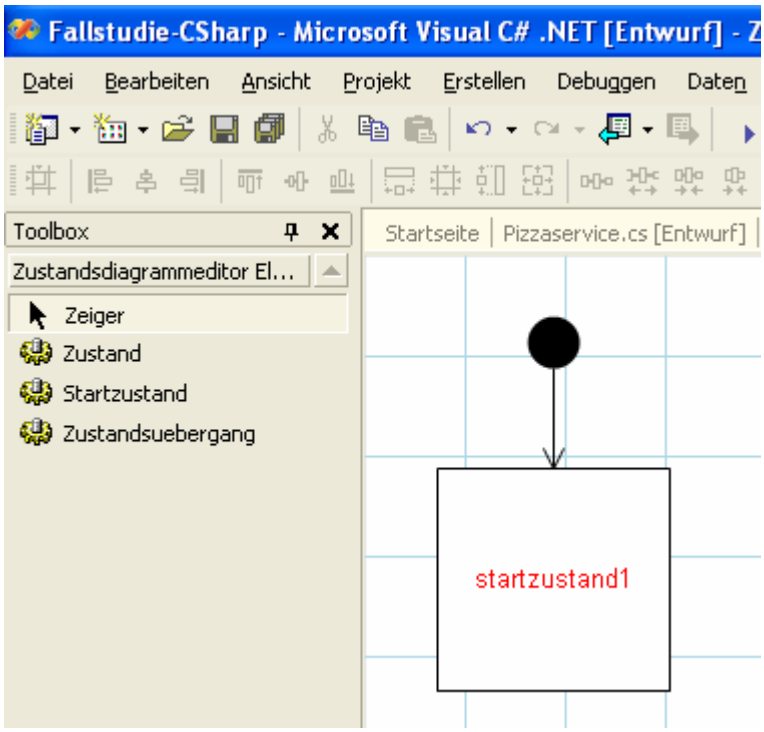
1. Wählen Sie aus der Toolbox unter der Kategorie „Zustandsdiagrammeditor Elemente“ das Element „Startzustand“ aus.



2. Ziehen Sie nun per Drag&Drop auf ihre Zeichenfläche einen Rahmen, der die Größe und Position ihres Startzustands beschreiben soll.



3. Beim Loslassen der Maus, also beim bestätigen des Drag&Drops, sollte ihr Startzustand erzeugt worden sein.



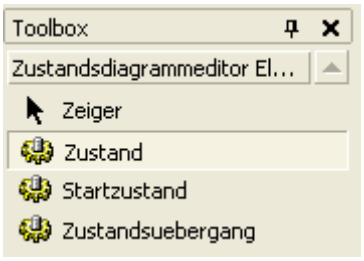
Alternativ können Sie ein Startzustand durch ein Doppelklick auf des Element „Startzustand“ in der Toolbox Kategorie „Zustandsdiagrammeditor Elemente“ erzeugen.

4.3 Erzeugung eines Zustands

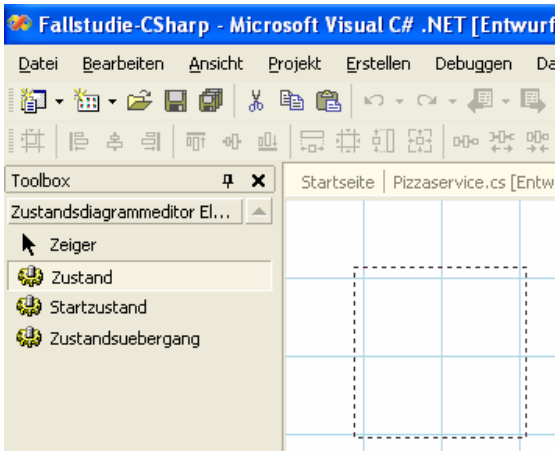
Bevor Sie einen Zustand erzeugen können, müssen Sie ein Zustandsdiagramm erzeugt haben (Siehe 4.1). Öffnen Sie dieses Zustandsdiagramm in seiner Designer-Ansicht.

Um nun einen Zustand per Drag&Drop zu erzeugen, gehen Sie wie folgt vor:

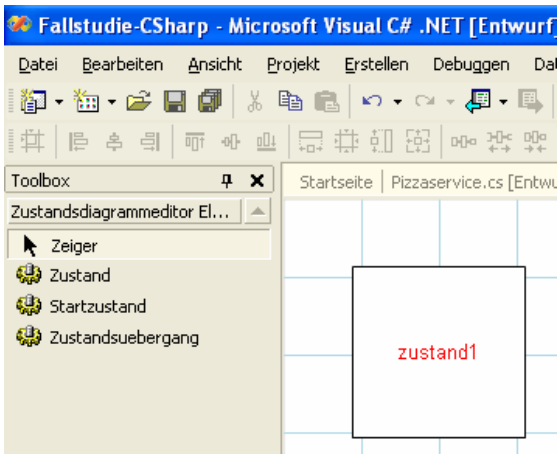
1. Wählen Sie aus der Toolbox unter der Kategorie „Zustandsdiagrammeditor Elemente“ das Element „Zustand“ aus.



2. Ziehen Sie nun per Drag&Drop auf ihre Zeichenfläche einen Rahmen, der die Größe und Position ihres Zustands beschreiben soll.



3. Beim Loslassen der Maus, also beim bestätigen des Drag&Drops, sollte ihr Zustand erzeugt worden sein.



Alternativ können Sie ein Zustand durch ein Doppelklick auf des Element „Zustand“ in der Toolbox Kategorie „Zustandsdiagrammeditor Elemente“ erzeugen.

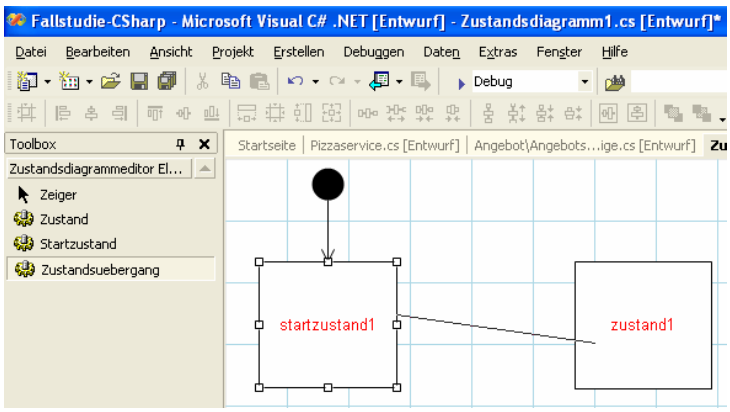
4.4 Erzeugung eines Zustandsübergang

Bevor Sie einen Zustandsübergang erzeugen können, müssen Sie ein Zustandsdiagramm erzeugt haben (Siehe 4.1). Öffnen Sie dieses Zustandsdiagramm in seiner Designer-Ansicht. Sie sollten außerdem wenigsten einen Zustand oder Startzustand erzeugt haben (siehe Abschnitt 4.2 und 4.3), um rekursive Zustandsübergänge und wenigsten zwei Zustände haben, um normale Zustandsübergänge erzeugen zu können.

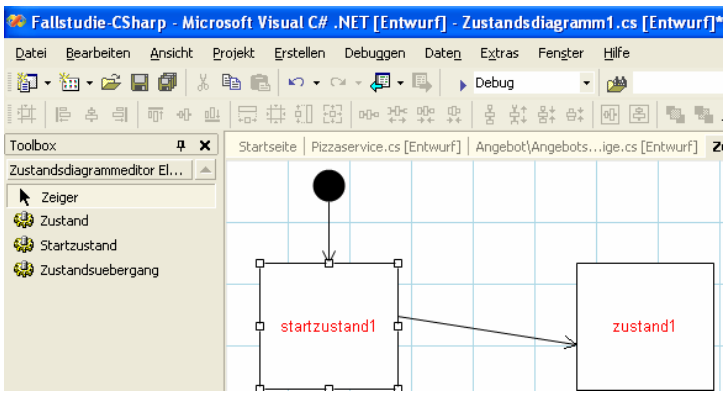
Um einen Zustandübergang zu erzeugen, gehen Sie wie folgt vor:

1. Wählen Sie aus der Toolbox unter der Kategorie „Zustandsdiagrammeditor Elemente“ das Element „Zustandsübergang“ aus.

2. Ziehen Sie nun per Drag&Drop auf ihre Zeichenfläche einen Zustandsübergang. Es kann sich hierbei um einen rekursiven oder normalen Zustandsübergang handeln. Der Zustandsübergang muss in die Pfeilrichtung gezogen werden.



3. Beim Loslassen der Maus, also beim bestätigen des Drag&Drops, sollte ihr Zustandsübergang erzeugt worden sein.



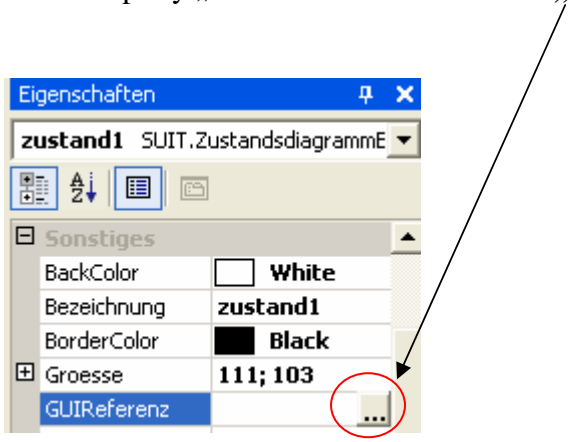
4.5 GUI-Maske Zuweisung zu einem Zustand

Um eine Maske einen Zustand (gilt auch für Startzustand) zuzuweisen, sollten Sie einen Zustandsdiagramm erzeugt und diesen in der Designer-Ansicht geöffnet haben (siehe Abschnitt 4.1). Außerdem sollte der zuzuweisende Zustand oder Startzustand erzeugt worden sein (siehe Abschnitt 4.2 und 4.3).

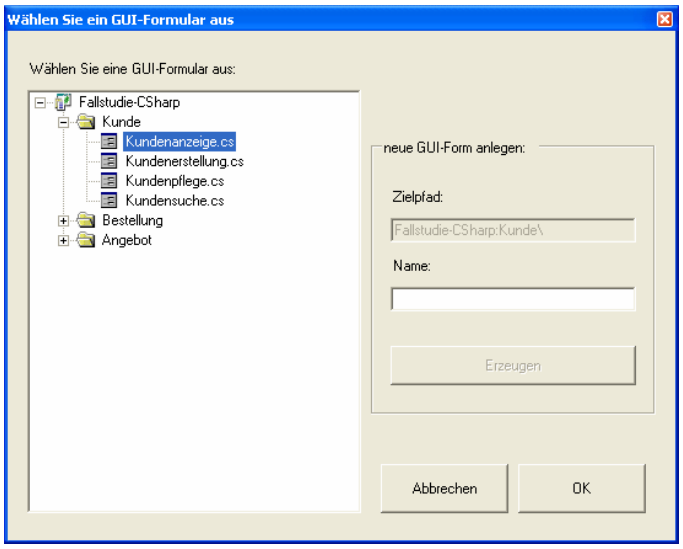
Um einen Zustand eine Maske zuzuweisen, gehen Sie wie folgt vor:

1. Markieren Sie ihren zuzuweisenden Zustand.

2. Doppelklicken Sie nun auf diesen Zustand, oder klicken Sie im Eigentschaftsfenster beim Property „GUIReferenz“ den Button **„...“** an.



3. Wählen Sie entweder aus der Liste eine bestehende GUI-Maske aus, und bestätigen dies mit dem Button „OK“ oder geben Sie unter dem Eingabefeld „Name“ den Namen für die zu erzeugende GUI-Maske ein. Achten sie dabei darauf, in welchen Pfad in der GUI-Auswahlliste sie sich gerade befinden. In diesem Pfad wird ihre Maske beim Erzeugen abgelegt. Wenn Sie den Namen der zu erzeugenden Maske eingeben haben, bestätigen Sie dies mit dem Button „Erzeugen“. Die GUI-Referenz ist danach dem Zustand zugewiesen.

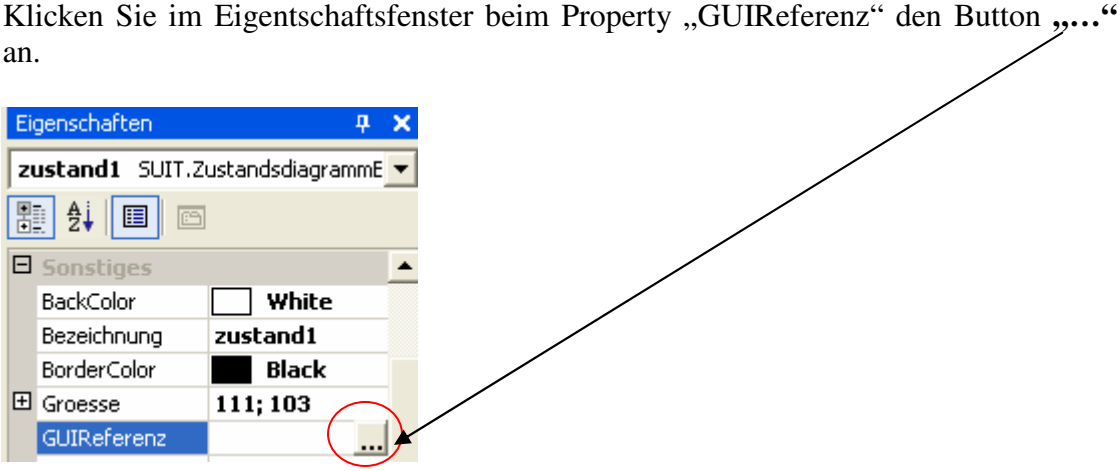


4.6 GUI-Maske Zuweisung eines Zustands ändern

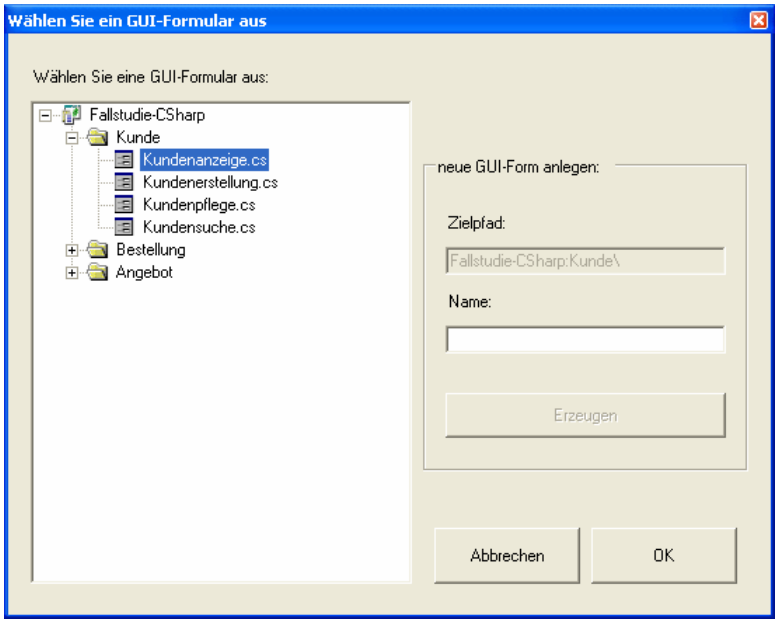
Um die GUI-Maske Zuweisung eines Zustands zu ändern, gehen Sie wie folgt vor:

1. Markieren Sie ihren zu ändernden Zustand.

2. Klicken Sie im Eigentschaftsfenster beim Property „GUIReferenz“ den Button **„…“** an.



3. Wählen Sie entweder aus der Liste eine bestehende GUI-Maske aus, und bestätigen dies mit dem Button „OK“ oder geben Sie unter dem Eingabefeld „Name“ den Namen für die zu neu erzeugende GUI-Maske ein. Achten sie dabei darauf, in Welchen Pfad in der GUI-Auswahlliste sie sich gerade befinden. In diesem Pfad wird ihre beim Erzeugen abgelegt. Wenn Sie den Namen des zu erzeugenen Maske eingeben haben, bestätigen Sie dies mit dem Button „Erzeugen“. Die neue GUI-Referenz ist danach dem Zustand zugewiesen.



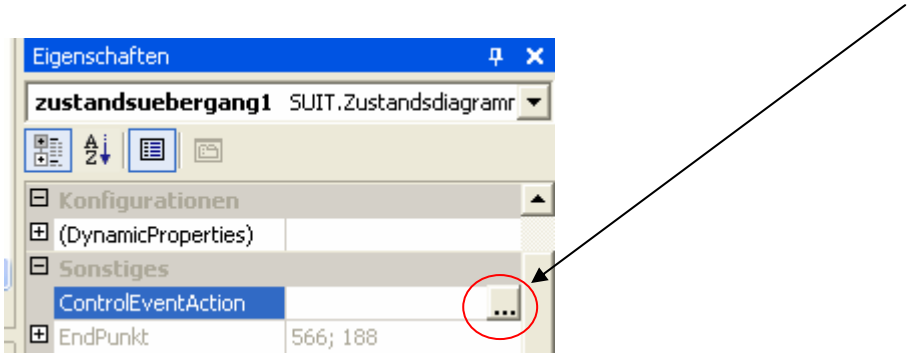
4.7 Zuweisung von Event und Action zu einem Zustandsübergang

Um Event und Action zu einem Zustandsübergang zuzuweisen, sollten Sie ein Zustandsdiagramm erzeugt und dieses in der Designer-Ansicht geöffnet haben (siehe Abschnitt 4.1). Außerdem sollte der zuzuweisende Zustand oder Startzustand erzeugt worden sein (siehe Abschnitt 4.4) und dem ausgehenden Zustand des Zustandsübergang eine GUI-Maske zugewiesen sein (siehe Abschnitt 4.5).

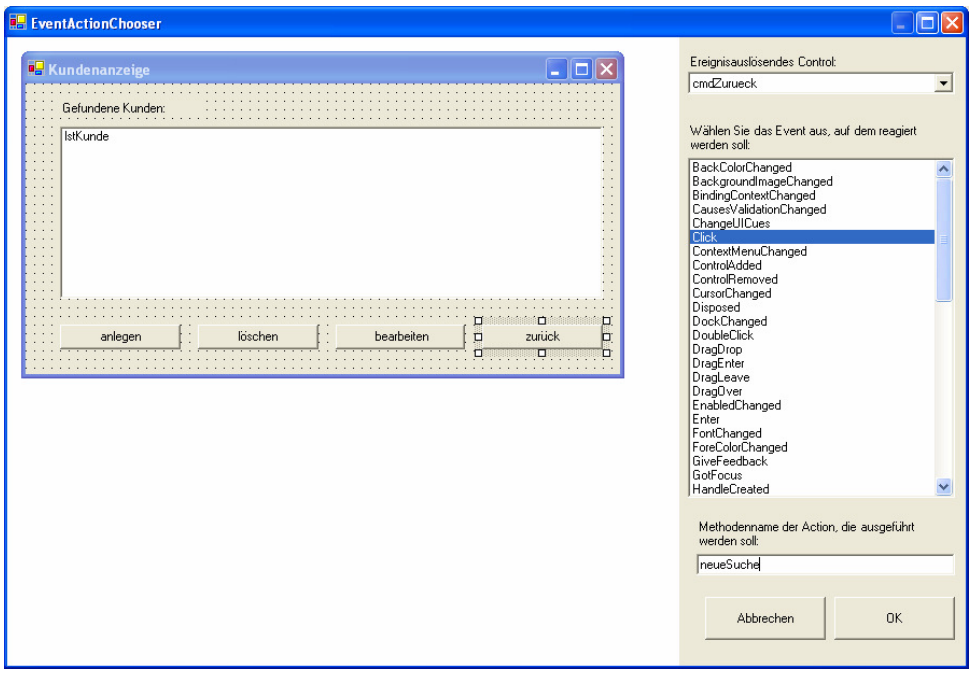
Um Event und Action zu einem Zustandsübergang zuzuweisen, gehen Sie wie folgt vor:

1. Markieren Sie den Zustandsübergang, der Event und Action zugewiesen bekommen soll.

2. Doppelklicken Sie nun auf diesen Zustandsübergang, oder klicken Sie im Eigenschaftsfenster beim Property „ControlEventAction“ den Button **„...“** an.

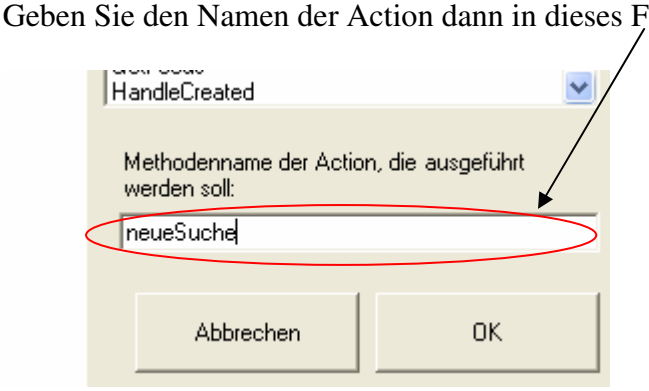


3. Ein Dialog-Fenster erscheint, in dem Sie die GUI-Maske, die Sie beim ausgehenden Zustand des Zustandsübergangs zugewiesen haben, sehen können. Je nachdem, wie ihre GUI-Maske aussieht, gibt es also hier Abweichung von den folgenden Screenshot:



4. Wählen Sie im ihre GUI-Masken-Ansicht (linkes Fensterbereich des Dialogs) das Control aus, auf dessen Ereignis reagiert werden soll, und wählen Sie dann das Ereignis aus der Liste rechts im Fenster aus.

5. Geben Sie den Namen der Action dann in dieses Feld ein.



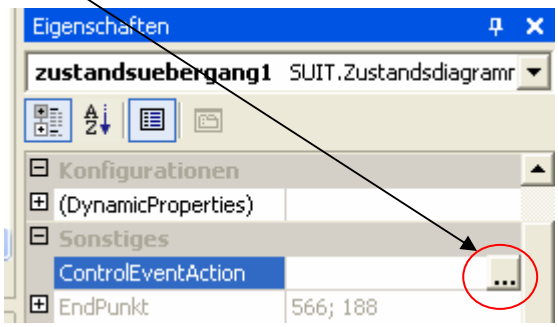
6. Bestätigen Sie nun mit dem Button „OK“. Ihr Zustandsübergang sollte nun erfolgreich das ausgewählte Event und Action zugewiesen sein.

4.8 Event und Action eines Zustandsübergang ändern

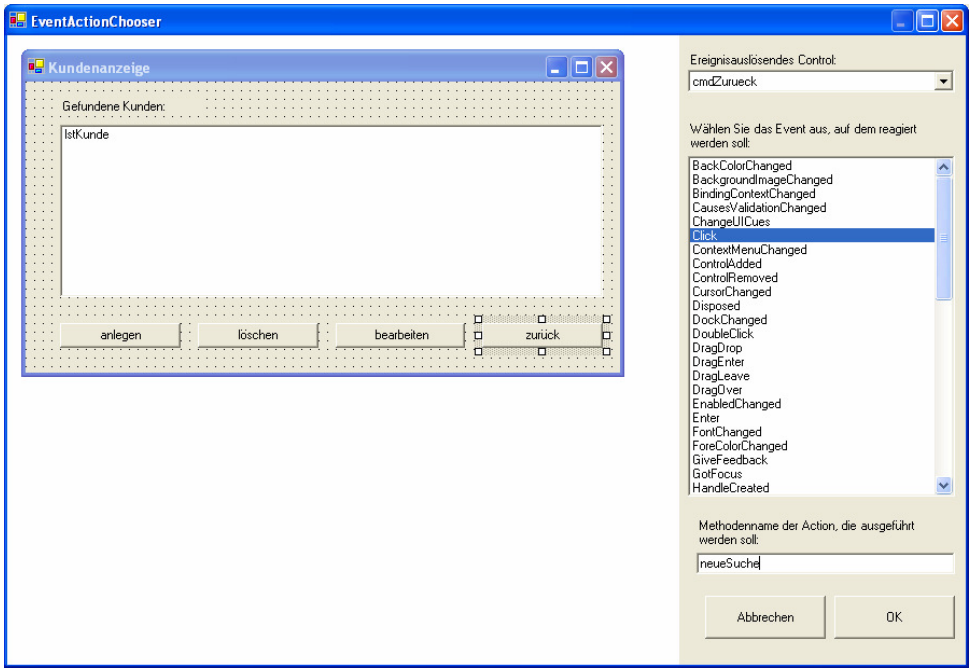
Um Event und Action eines Zustandsübergangs zu ändern, gehen Sie wie folgt vor:

1. Markieren Sie den Zustandsübergang, dessen Event und Action geändert werden soll.

2. Klicken Sie im Eigenschaftsfenster beim Property „ControlEventAction“ den Button **„...“** an.

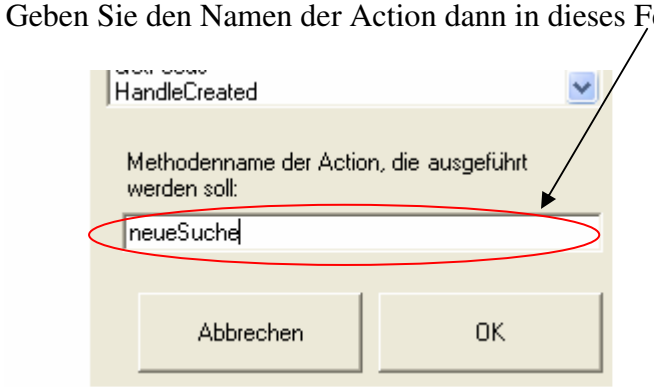


3. Ein Dialog-Fenster erscheint, in dem Sie die GUI-Maske, die Sie beim ausgehenden Zustand des Zustandsübergangs zugewiesen haben, sehen können. Je nachdem, wie ihre GUI-Maske aussieht, gibt es also hier Abweichung von den folgenden Screenshot:



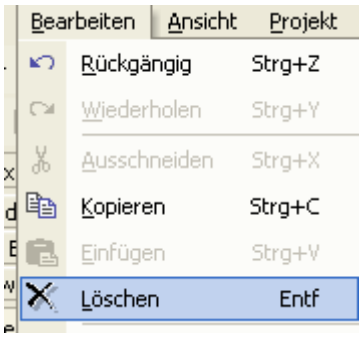
4. Wählen Sie im ihre GUI-Masken-Ansicht (linkes Fensterbereich des Dialogs) das Control aus, auf dessen Ereignis reagiert werden soll, und wählen Sie dann das Ereignis aus der Liste rechts im Fenster aus.

5. Geben Sie den Namen der Action dann in dieses Feld ein.



6. Bestätigen Sie nun mit dem Button „OK“. Ihr Zustandsübergang sollte nun erfolgreich den neuen Event und Action zugewiesen sein.

4.9 Löschen von Elementen aus dem Zustandsdiagramm



Markieren Sie das zu löschenden Element im Zustandsdiagramm-Designer, und drücken Sie entweder die Taste „entf“ oder wählen das Menü: **Bearbeiten → Löschen** aus.

5. Fassadengenerierung

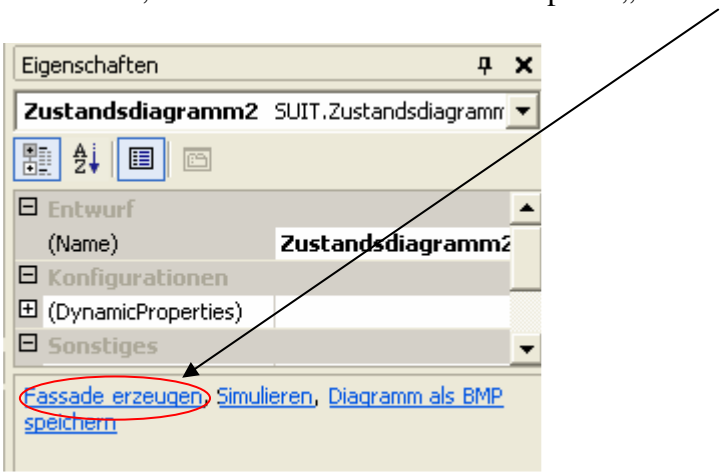
Bevor die Fassaden-Klasse generiert werden kann, muss ein Zustandsdiagramm erzeugt und in der Design-Ansicht geöffnet werden (siehe Abschnitt 4.1).

ACHTUNG: Bedenken Sie jedoch, dass die Fassadengenierung in der vorliegenden SUIT Version **nur C# Projekte** unterstützt.

Um eine Fassaden-Klasse zu genieren, gehen Sie wie folgt vor:

1. Markieren Sie das Zustandsdiagramm, in dem Sie auf eine freie Stelle der Zeichenfläche des Zustandsdiagramm-Designers anklicken.

2. Im Eigenschaftsfenster sollte nun die Eigenschaften des Zustandsdiagramms erscheinen, und unterhalb der Liste eine Option „Fassade erzeugen“ zu finden sein.



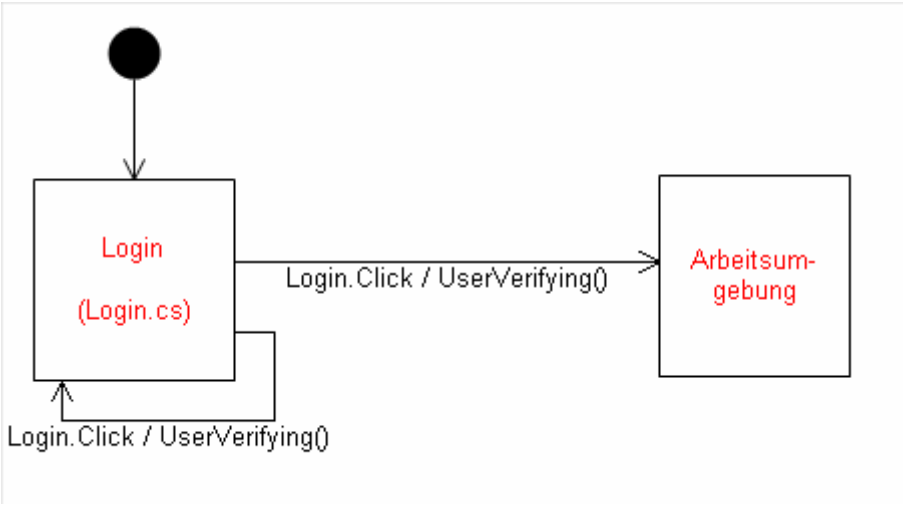
3. Klicken Sie auf die Option „Fassade erzeugen“. In Ihrem Projekt sollte nun eine Klasse „Fassade“ zu finden sein.

Tipp: *Wenn Sie bereits eine Fassade erzeugt haben, wird diese Fassade mit den hier beschriebenen Schritten 1-3 wieder aktualisiert.*

6. Simulation

6.1 Voraussetzung für die Simulation

Bevor Sie simulieren, sollten Sie das Zustandsdiagramm erstellt (siehe Abschnitt 4.1) und ihr Modell entworfen haben. Ihr Modell muss über einen Startzustand verfügen, alle erreichbaren Zustände sollten eine gültige GUI Maske zugewiesen bekommen haben (Siehe Abschnitt 4.5) und **vermeiden** Sie in Ihrem Modell unklare Wegverzweigungen, wie z.B. in der unteren Abbildung, wo ein Ereignis zu zwei unterschiedliche Zustände führt.

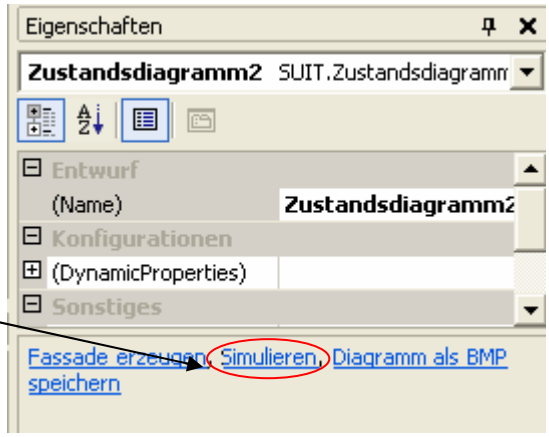


Bevor die Simulation ausgeführt werden kann, sollten Sie die Fassaden-Klasse genieren lassen und dafür sorgen, dass Ihr Projekt fehlerfrei kompilierbar ist.

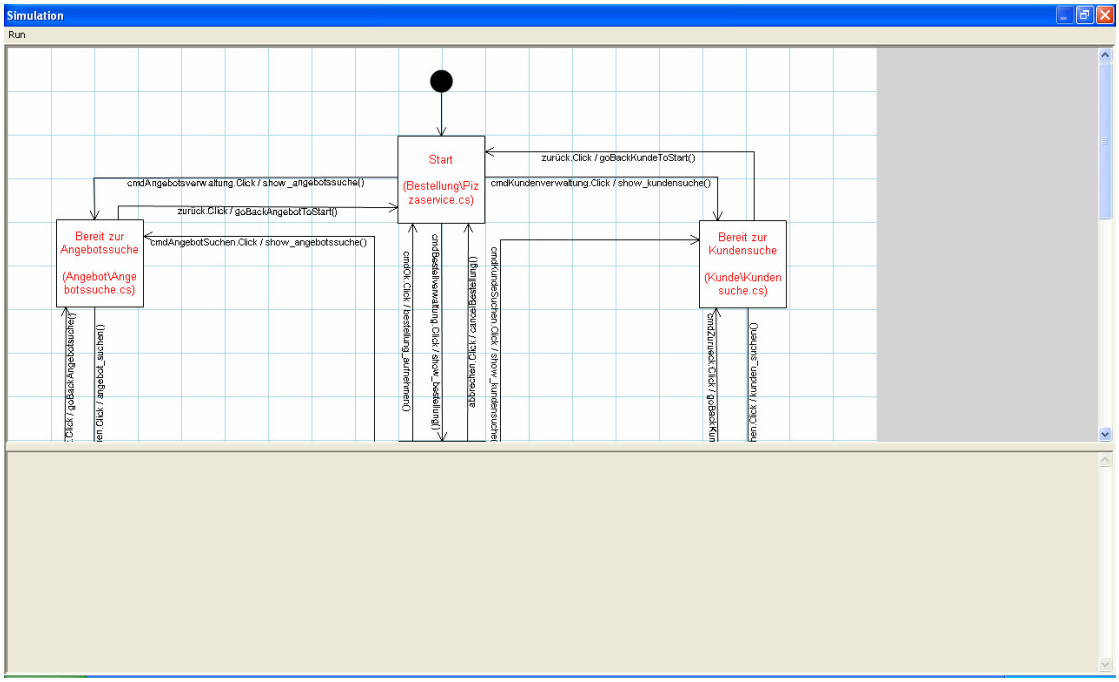
6.2 Wie startet man eine Simulation

Um eine Simulation zu starten, gehen Sie wie folgt vor:

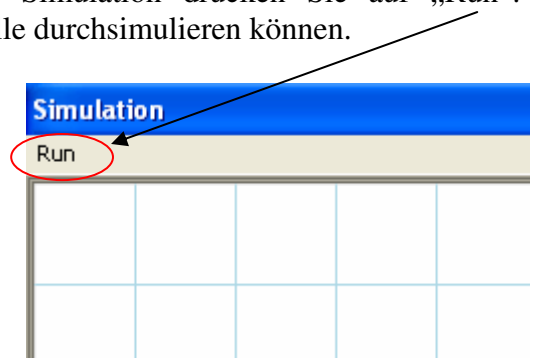
1. Markieren Sie das Zustandsdiagramm, in dem Sie auf eine freie Stelle der Zeichenfläche des Zustandsdiagramm-Designers anklicken.
2. Im Eigenschaftsfenster sollte nun die Eigenschaften des Zustandsdiagramms erscheinen, und unterhalb der Liste eine Option „Fassade erzeugen“ zu finden sein.
3. Klicken Sie auf die Option „Simulieren“.



4. Sie sollten nun das Simulationsfenster sehen, in der Sie ihr Zustandsdiagramm-Modell zu sehen ist.

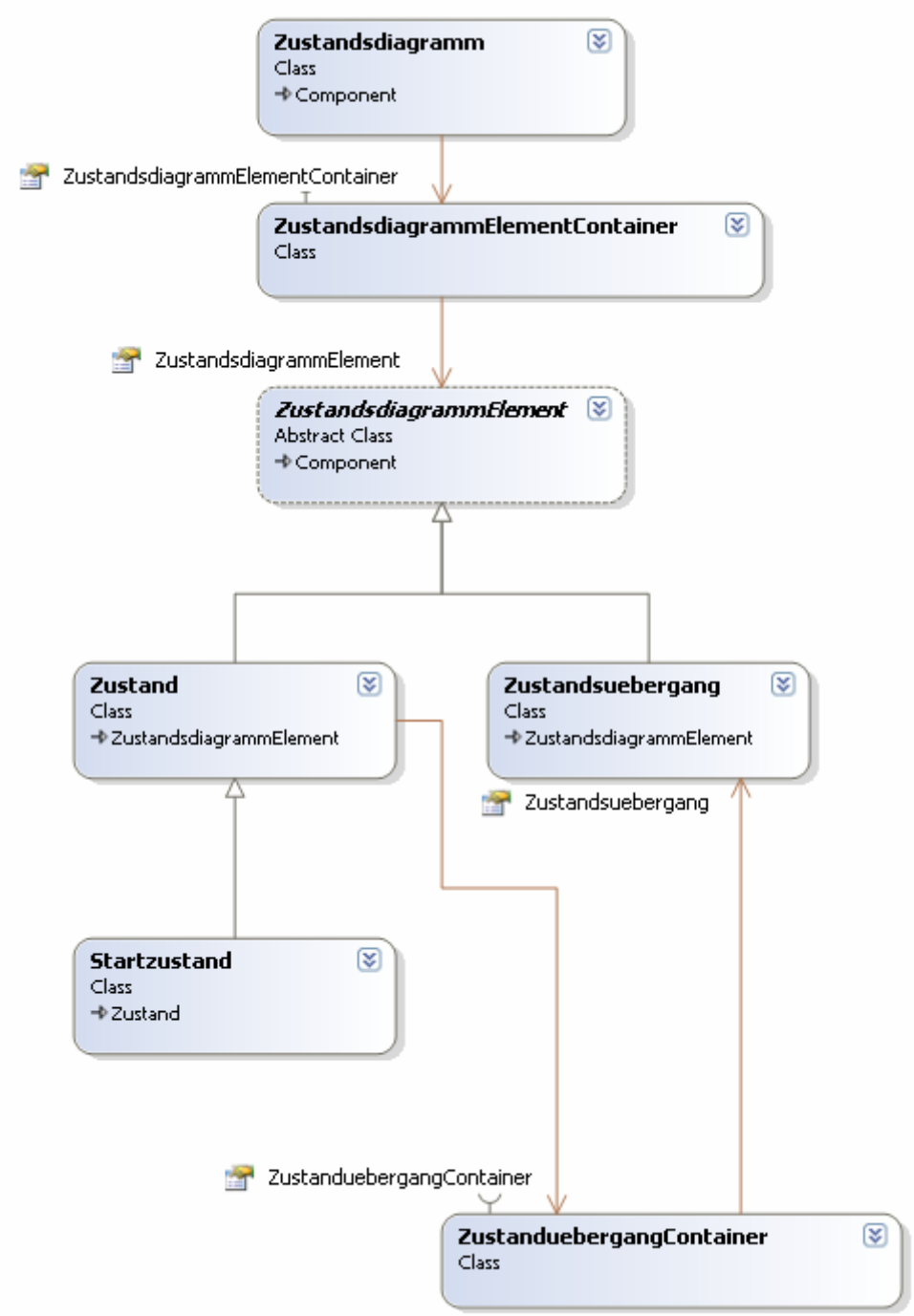


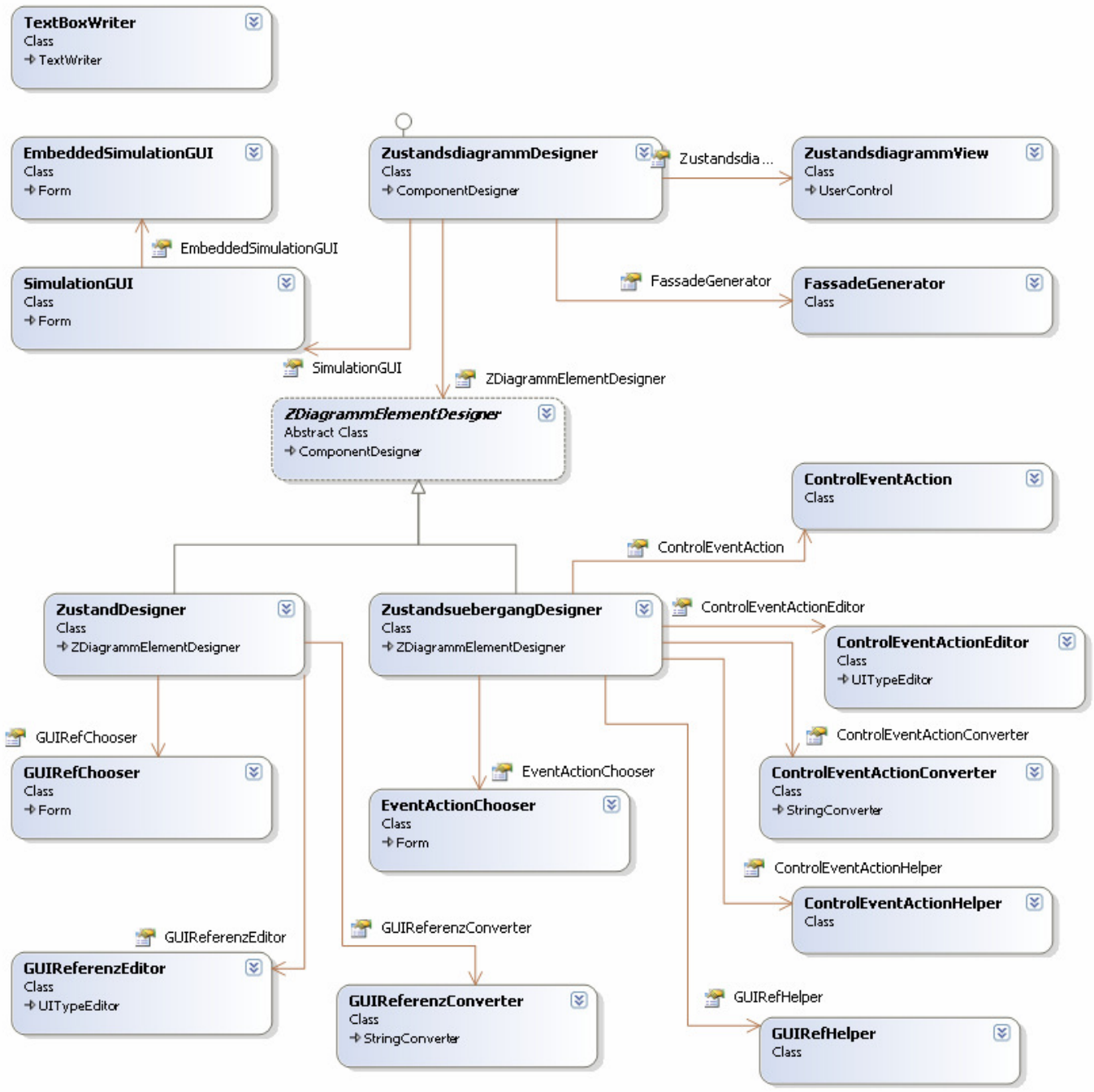
5. Zum Ausführen der Simulation drücken Sie auf „Run“. Nun sollten Sie ihre Benutzungsschnittstelle durchsimulieren können.



Tipp: *Sie können das Simulationsfenster während der Simulation schließen, ohne das die Simulation abgebrochen wird. So können Sie dem Kunden ein Prototyp vorführen, ohne Ihn mit dem Zustandsdiagrammmodell zu konfrontieren...*

Klassendiagramm




TextBoxWriter
Class
TextWriter
EmbeddedSimulationGUI
Class
Form
ZustandsdiagrammDesigner
Class
ComponentDesigner
Zustandsdia...
ZustandsdiagrammView
Class
UserControl
EmbeddedSimulationGUI
SimulationGUI
Class
Form
FassadeGenerator
FassadeGenerator
Class
SimulationGUI
ZDiagrammElementDesigner
ZDiagrammElementDesigner
Abstract Class
ComponentDesigner
ControlEventAction
Class
ControlEventAction
ZustandDesigner
Class
ZDiagrammElementDesigner
ZustandsuebergangDesigner
Class
ZDiagrammElementDesigner
ControlEventActionEditor
ControlEventActionEditor
Class
UITypeEditor
GUIRefChooser
GUIRefChooser
Class
Form
ControlEventActionConverter
ControlEventActionConverter
Class
StringConverter
EventActionChooser
EventActionChooser
Class
Form
ControlEventActionHelper
ControlEventActionHelper
Class
GUIReferenzEditor
GUIReferenzEditor
Class
UITypeEditor
GUIReferenzConverter
GUIReferenzConverter
Class
StringConverter
GUIRefHelper
GUIRefHelper
Class